File - Field.h

```cpp
#ifndef BUFFY_FIELD_H
#define BUFFY_FIELD_H

#include <list>
#include <memory>
#include "Humanoid.h"

class Humanoid;

/**
 * This class represents the Field of the simulation.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Field {
public:

    /**
     * Constructor for a new Field.
     *
     * @param width     the width of the actual field
     * @param height    the height of the actual field
     * @param nbVampire the number of Vampire at the start of the game
     * @param nbHuman   the number of Human at the start of the game
     */
    Field(size_t width, size_t height, size_t nbVampire, size_t nbHuman);

    /**
     * Advance the simulation of one turn (action).
     *
     * @return  true if there's still humans or vampires in the field, false otherwise, as the game is over.
     */
    bool nextTurn();

    /**
     * Get the closest Humanoïd of type given from the humanoïd given
     *
     * @param from  the humanoïd from which we want to search
     * @param type  the type of humanoïd we research
     * @return      the closest humanoïd from the given one
     */
    [[nodiscard]] std::shared_ptr<Humanoid> getClosest(const std::shared_ptr<Humanoid>& from, Humanoid::Type type) const;

    /**
     * Get the max width of the field
     * @return
     */
    [[nodiscard]] size_t getWidth() const;

    /**
     * Get the max height of the field
     * @return
     */
    [[nodiscard]] size_t getHeight() const;

    /**
     * Get the number of Vampire in the field
     * @return
     */
    [[nodiscard]] size_t getNbVampire() const;

    /**
    * Get the number of Human in the field
    * @return
    */
    [[nodiscard]] size_t getNbHuman() const;

    /**
     * Get the current turn of the simulation
     * @return
     */
    [[nodiscard]] int getTurn() const;

    /**
     * Add a humanoïd to our simulation.
     *
     * @param hum   the humanoïd we want to add to our simulation
     */
```

File - Field.h

```
    void addHumanoid(const std::shared_ptr<Humanoid>& hum);

    /**
     * Increment our number of Vampire in our simulation.
     */
    void incrementVampireCount();


    void operator=(const Field&) = delete;

    /**
     * Get the list of humanoïds
     * @return
     */
    [[nodiscard]] const std::list<std::shared_ptr<Humanoid>>& getHumanoids() const;

private:
    std::list<std::shared_ptr<Humanoid>> _humanoids;
    int _turn;
    size_t _width;
    size_t _height;
    size_t _nbVampire;
    size_t _nbHuman;
};
#include "Humanoid.h"

#endif  // BUFFY_FIELD_H

```

File - Action.h

```cpp
#ifndef BUFFY_ACTION_H
#define BUFFY_ACTION_H

#include <cstddef>
#include "Humanoid.h"
#include "utilities/Point.h"
class Field;

/**
 * This class represents an Action.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Action {
protected:
    // The humanoïd on which we want to apply the action
    std::weak_ptr<Humanoid> _target;

public:
    /**
     * Executes the action.
     *
     * @param f the field of the simulation
     */
    virtual void execute(Field& f) = 0;

    /**
     * Virtual destructor.
     */
    virtual ~Action() = default;

    /**
     * Constructor for a new Action.
     *
     * @param target the humanoïd on which we want to apply the action
     */
    explicit Action(const std::weak_ptr<Humanoid>& target);
};

#include "Field.h"

#endif  // BUFFY_ACTION_H

```

File - main.cpp

```cpp
#include <cstdlib>
#include <functional>
#include <iomanip>
#include <iostream>
#include <map>
#include "Field.h"
#include "Renderer.h"

using namespace std;

/**
 * Main function of our simulation, handles the launch arguments,
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
int main(int argc, char** argv) {
    // Game has default values of 50x50, 10 vampires and 20 humans
    size_t width = 50;
    size_t height = 50;
    size_t nbVampire = 10;
    size_t nbHuman = 20;

    if (argc >= 5) {
        width = atoll(argv[1]);
        height = atoll(argv[2]);
        nbVampire = atoll(argv[3]);
        nbHuman = atoll(argv[4]);
    }

    Field field(width, height, nbVampire, nbHuman);
    Renderer::print(field);

    bool quit = false;
    std::map<char, function<void()>> actions = {
        {'q', [&quit]() { quit = true; }}, {'s', [width,height,nbVampire,nbHuman]() { Renderer::runSimulationsAndPrintStats
(width,height,nbVampire,nbHuman); }}, {'n', [&field]() {
                                                                                                                field.nextTurn();
                                                                                                                Renderer::print(
field);
                                                                                                            }}};

    string action;
    while (!quit) {
        cout << "[" << field.getTurn() << "] q> quit s>tatistics n>ext: ";
        cin >> action;
        char id = action.at(0);
        if (actions.find(id) == actions.end()) {
            cout << "Invalid action" << endl;
            continue;
        }

        actions.at(id)();
    }
}

// Check de type OK ici pour vérifier si c'est un vampire ou non !
```

File - Field.cpp

```cpp
#include "Field.h"
#include <list>
#include "humanoids/Buffy.h"
#include "humanoids/Human.h"
#include "humanoids/Vampire.h"
#include "utilities/Random.h"

const std::list<std::shared_ptr<Humanoid>>& Field::getHumanoids() const {
    return _humanoids;
}
Field::Field(size_t width, size_t height, size_t nbVampire, size_t nbHuman) {

    _turn = 0;
    _width = width;
    _height = height;
    _nbHuman = nbHuman;
    _nbVampire = nbVampire;

    for (size_t i = 0; i < nbHuman; ++i) {
        _humanoids.emplace_back(std::make_shared<Human>(width, height));
    }
    for (size_t i = 0; i < nbVampire; ++i) {
        _humanoids.emplace_back(std::make_shared<Vampire>(width, height));
    }
    _humanoids.emplace_back(std::make_shared<Buffy>(width, height));
}

bool Field::nextTurn() {
    // Déterminer les prochaines actions
    for (std::list<std::shared_ptr<Humanoid>>::iterator it = _humanoids.begin(); it != _humanoids.end(); it++)
        (*it)->setAction(*this);
    // Executer les actions
    for (std::list<std::shared_ptr<Humanoid>>::iterator it = _humanoids.begin(); it != _humanoids.end(); it++)
        (*it)->executeAction(*this);
    // Enlever les humanoides tués
    for (std::list<std::shared_ptr<Humanoid>>::iterator it = _humanoids.begin(); it != _humanoids.end();) {
        if (!(*it)->isAlive()) {
            std::shared_ptr<Humanoid> toDelete = *it;
            it = _humanoids.erase(it);  // suppression de l'élément dans la liste

            if (toDelete->getType() == Humanoid::Type::VAMPIRE){
                --_nbVampire;
            }
            else if (toDelete->getType() == Humanoid::Type::HUMAN){
                --_nbHuman;
            }
            toDelete.reset(); // destruction de l'humanoide référencé
        } else
            ++it;
    }
    ++_turn;

    return _nbVampire != 0;
}



std::shared_ptr<Humanoid> Field::getClosest(const std::shared_ptr<Humanoid>& from, Humanoid::Type type) const {
    Point posFrom = from->getPosition();
    size_t minDist = std::numeric_limits<size_t>::max();
    std::shared_ptr<Humanoid> minHum;

    for (const auto& hum : _humanoids) {
        if(hum == from)
            continue;
        if (hum->getType() == type) {
            size_t humDist = posFrom.distance(hum->getPosition());
            if (humDist < minDist) {
                minHum = hum;
                minDist = humDist;
            }
        }
    }
    return minHum;
}

void Field::incrementVampireCount() {
    ++_nbVampire;
}

```

File - Field.cpp

```cpp
size_t Field::getWidth() const {
    return _width;
}

size_t Field::getHeight() const {
    return _height;
}
void Field::addHumanoid(const std::shared_ptr<Humanoid>& hum) {
    _humanoids.push_back(hum);
}

size_t Field::getNbVampire() const {
    return _nbVampire;
}

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

int Field::getTurn() const {
    return _turn;
}


```

File - Action.cpp

```cpp
#include "Action.h"


Action::Action(const std::weak_ptr<Humanoid>& target) : _target(target) {}
```

File - Humanoid.h

```cpp
#ifndef BUFFY_HUMANOID_H
#define BUFFY_HUMANOID_H

#include <cstddef>
#include <memory>
#include "utilities/Point.h"

class Field;
class Action;

/**
 * @class Humanoid
 * @brief A class representing a humanoid entity in the game.
 *
 * This class is a base class for different types of humanoid entities in the game.
 * It provides the basic functionalities that all humanoid entities should have.
 *
 * @note This class is designed to be inherited by other classes.
 */
class Humanoid : public std::enable_shared_from_this<Humanoid> {
public:

    /**
     * @class Iterator
     * @brief An iterator class for iterating over Humanoid objects.
     */
    struct Iterator {
        explicit Iterator(Humanoid* ptr) : _ptr(ptr) {}

        Humanoid& operator*() const { return *_ptr; };
        Humanoid* operator->() { return _ptr; };

        Iterator& operator++() {
            _ptr++;
            return *this;
        }
        Iterator operator++(int) {
            Iterator tmp = *this;
            ++(*this);
            return tmp;
        }

        friend bool operator==(const Iterator& a, const Iterator& b) { return a._ptr == b._ptr; };
        friend bool operator!=(const Iterator& a, const Iterator& b) { return a._ptr != b._ptr; };

    private:
        Humanoid* _ptr;
    };

    /**
     * @enum Type
     * @brief An enumeration of the different types of humanoid entities.
     */
    enum class Type { HUMAN, VAMPIRE, BUFFY };

    Humanoid(size_t maxX, size_t maxY);
    explicit Humanoid(const Point& pos);
    virtual ~Humanoid() = default;

    /**
     * @brief A pure virtual function for setting the action of the humanoid.
     *
     * @param f The field on which the action is to be set.
     */
    virtual void setAction(Field& f) = 0;

    /**
     * @brief Executes the action of the humanoid on a given field.
     *
     * @param f The field on which the action is to be executed.
     */
    void executeAction(Field& f);

    /**
     * @brief Sets the position of the humanoid.
     *
     * @param p The new position of the humanoid.
     */
    void setPosition(Point p);

```

File - Humanoid.h

```
81      /**
82       * @brief Gets the position of the humanoid.
83       *
84       * @return The position of the humanoid.
85       */
86      [[nodiscard]] Point getPosition() const;
87
88      /**
89       * @brief Checks if the humanoid is alive.
90       *
91       * @return True if the humanoid is alive, false otherwise.
92       */
93      [[nodiscard]] bool isAlive() const;
94
95      /**
96       * @brief A pure virtual function for getting the type of the humanoid.
97       *
98       * @return The type of the humanoid.
99       */
100     [[nodiscard]] virtual Type getType() const = 0;
101
102     /**
103      * @brief A pure virtual function for getting the amount of movement of the humanoid.
104      *
105      * @return The amount of movement of the humanoid.
106      */
107     [[nodiscard]] virtual std::size_t getMoveAmount() const = 0;
108
109     /**
110      * @brief Marks the humanoid as dead.
111      */
112     void dies();
113
114 protected:
115     Point _position;
116     bool _alive;
117     std::shared_ptr<Action> _action;
118 };
119
120 #include "Action.h"
121 #include "Field.h"
122
123 #endif  // BUFFY_HUMANOID_H
124
```

File - Renderer.h

```cpp
#ifndef BUFFY_RENDERER_H
#define BUFFY_RENDERER_H
#include <cstddef>
#include <map>
#include "Humanoid.h"

const char HORIZONTAL_BORDER = '-';
const char VERTICAL_BORDER = '|';
const size_t NUMBER_OF_SIMULATIONS = 10000;

// Constants for the representation of the game field
static const std::map<Humanoid::Type, char> LETTERS = {
    {Humanoid::Type::BUFFY, 'B'},
    {Humanoid::Type::HUMAN, 'h'},
    {Humanoid::Type::VAMPIRE, 'v'},
};


/**
 * This class represents the Renderer that allow us to run the game and its logic.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Renderer {
public:
    /**
     * @brief Runs a number of simulations and prints the statistics.
     *
     * This method runs a number of simulations and prints the win statistics.
     * The number of simulations is defined by the constant NUMBER_OF_SIMULATIONS.
     *
     * @param width The width of the game field.
     * @param height The height of the game field.
     * @param nbVampire The number of vampires in the game.
     * @param nbHuman The number of humans in the game.
     */
    static void runSimulationsAndPrintStats(size_t width, size_t height, size_t nbVampire, size_t nbHuman);

    /**
     * @brief Prints the game field.
     *
     * This method prints the game field. It uses the LETTERS map to represent each type of humanoid.
     *
     * @param field The game field to print.
     */
    static void print(const Field& field);

    /**
     * @brief Prints a line of a specified length.
     *
     * This method prints a line of a specified length. It uses the HORIZONTAL_BORDER character for the line.
     *
     * @param length The length of the line to print.
     */
    static void printLine(size_t length);
};

#endif  // BUFFY_RENDERER_H

```

File - Humanoid.cpp

```cpp
#include "Humanoid.h"
#include "utilities/Random.h"

Humanoid::Humanoid(size_t maxX, size_t maxY) : _position(Random::getRandomPos(0, maxX - 1, 0, maxY - 1)), _alive(true) {}
Humanoid::Humanoid(const Point& pos) : _position(pos), _alive(true) {}

Point Humanoid::getPosition() const {
    return _position;
}

void Humanoid::setPosition(Point position) {
    _position = position;
}

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

void Humanoid::dies() {
    _alive = false;
}

void Humanoid::executeAction(Field& f) {
    if (isAlive()) {
        if (_action)
            _action->execute(f);
    }
}

```

File - Renderer.cpp

```cpp
#include "Renderer.h"
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
#include "Field.h"
#include "Humanoid.h"
#include "cstring"

void Renderer::runSimulationsAndPrintStats(size_t width, size_t height, size_t nbVampire, size_t nbHuman) {
    size_t nbWin = 0;
    for (size_t i = 0; i < NUMBER_OF_SIMULATIONS; ++i) {
        Field f(width, height, nbVampire, nbHuman);
        while (f.nextTurn()) {
            // Do nothing since the logic is done in the nextTurn method
        }
        if (f.getNbHuman()) { // If there are still humans alive
            ++nbWin;
        }
    }

    std::cout << std::fixed << std::setprecision(2);  // Set output format
    std::cout << "Win rate: " << static_cast<double>((double)nbWin / (double)NUMBER_OF_SIMULATIONS * 100) << "%" << std::endl;
    // Reset output format
    std::cout.unsetf(std::ios_base::fixed);
    std::cout << std::setprecision(6);
}

void Renderer::print(const Field& field) {
    printLine(field.getWidth() + 2);
    std::vector<std::vector<char>> grid(field.getHeight(), std::vector<char>(field.getWidth(), ' '));

    // Building grid
    for (const auto& humanoid : field.getHumanoids()) {
        auto position = humanoid->getPosition();
        grid.at(position.y).at(position.x) = LETTERS.at(humanoid->getType());
    }

    // Printing full grid
    std::string gridAsText;
    for (const auto& line : grid) {
        gridAsText += VERTICAL_BORDER;
        for (const auto cell : line) {
            gridAsText += cell;
        }
        gridAsText += VERTICAL_BORDER;
        gridAsText += "\n";
    }
    std::cout << gridAsText;
    printLine(field.getWidth() + 2);
}

void Renderer::printLine(size_t length) {
    std::string line;
    for (size_t i = 0; i < length; ++i) {
        line += HORIZONTAL_BORDER;
    }
    std::cout << line << std::endl;
}

```

File - Buffy.h

```cpp
#ifndef BUFFY_BUFFY_H
#define BUFFY_BUFFY_H

#include "KillerHumanoid.h"

/**
 * This class represents Buffy.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Buffy : public KillerHumanoid {

public:
    /**
     * Constructor for a new Buffy
     *
     * @param maxX  maximum value of x position
     * @param maxY  maximum value of y position
     */
    Buffy(size_t maxX, size_t maxY);

    /**
     * The type of Buffy in enum Humanoid::Type
     *
     * @return type BUFFY from enum Humanoid::Type
     */
    [[nodiscard]] Type getType() const override;

    /**
     * The number of cells it moves
     *
     * @return number of cells each move
     */
    [[nodiscard]] std::size_t getMoveAmount() const override;

    /**
     * Set the action of Buffy given the context (field)
     *
     * @param f the field of the simulation
     */
    void setAction(Field &f) override;

};

#endif  // BUFFY_BUFFY_H

```

File - Human.h

```cpp
#ifndef BUFFY_HUMAN_H
#define BUFFY_HUMAN_H

#include "../Humanoid.h"

/**
 * This class represents a Human.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Human : public Humanoid {

public:

    /**
     * Constructor for a new Human
     *
     * @param maxX  maximum value of x position
     * @param maxY  maximum value of y position
     */
    Human(size_t maxX, size_t maxY);

    /**
     * The type of Human in enum Humanoid::Type
     *
     * @return type HUMAN from enum Humanoid::Type
     */
    Type getType() const override;

    /**
     * The number of cells it moves
     *
     * @return number of cells each move
     */
    std::size_t getMoveAmount() const override;

    /**
     * Set the action of our Human given the context (field)
     *
     * @param f the field of the simulation
     */
    void setAction(Field &f) override;

};

#endif  // BUFFY_HUMAN_H

```

File - Buffy.cpp

```cpp
#include "Buffy.h"
#include "../subActions/ChaseAction.h"
#include "../subActions/KillAction.h"
#include "../subActions/MoveRandomAction.h"

Buffy::Buffy(size_t maxX, size_t maxY) : KillerHumanoid(maxX, maxY) {}

Humanoid::Type Buffy::getType() const {
    return Humanoid::Type::BUFFY;
}

std::size_t Buffy::getMoveAmount() const {
    return 2;
}
void Buffy::setAction(Field& f) {
    if (!f.getNbVampire()) {
        _action = std::make_unique<MoveRandomAction>(std::weak_ptr<Humanoid>(shared_from_this()));
    } else {
        std::pair<std::shared_ptr<Humanoid>, bool> vampire = findHumanoidToAffect(f, Humanoid::Type::VAMPIRE);
        if (vampire.second) {
            _action = std::make_unique<KillAction>(vampire.first);
        } else {
            _action = std::make_unique<ChaseAction>(std::weak_ptr<Humanoid>(shared_from_this()), vampire.first);
        }
    }
}

```

File - Human.cpp

```cpp
#include "Human.h"
#include "../subActions/MoveRandomAction.h"

Human::Human(size_t maxX, size_t maxY) : Humanoid(maxX, maxY) {}

Humanoid::Type Human::getType() const {
    return Humanoid::Type::HUMAN;
}

std::size_t Human::getMoveAmount() const {
    return 1;
}
void Human::setAction(Field& f) {
    _action = std::make_unique<MoveRandomAction>(std::weak_ptr<Humanoid>(shared_from_this()));
}

```

File - Vampire.h

```h
#ifndef BUFFY_VAMPIRE_H
#define BUFFY_VAMPIRE_H

#include "KillerHumanoid.h"

/**
 * This class represents a Vampire.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Vampire : public KillerHumanoid {

public:
    /**
     * Constructor for a new Vampire.
     *
     * @param maxX  maximum value of it's x position
     * @param maxY  maximum value of it's y position
     */
    Vampire(size_t maxX, size_t maxY);

    /**
     * Constructor for a new Vampire.
     *
     * @param pos   the position of our new Vampire
     */
    explicit Vampire(Point pos);

    /**
     * The type of Vampire in enum Humanoid::Type
     *
     * @return type VAMPIRE from enum Humanoid::Type
     */
    Type getType() const override;

    /**
     * The number of cells it moves
     *
     * @return number of cells each move
     */
    std::size_t getMoveAmount() const override;

    /**
     * Set the action of our Vampire given the context (field)
     *
     * @param f the field of the simulation
     */
    void setAction(Field& f) override;


};

#endif  // BUFFY_VAMPIRE_H

```

File - Vampire.cpp

```cpp
#include "Vampire.h"
#include <iostream>
#include "../subActions/BiteAction.h"
#include "../subActions/ChaseAction.h"

Vampire::Vampire(size_t maxX, size_t maxY) : KillerHumanoid(maxX, maxY) {}

Vampire::Vampire(Point pos) : KillerHumanoid(pos) {}

Humanoid::Type Vampire::getType() const {
    return Humanoid::Type::VAMPIRE;
}
void Vampire::setAction(Field& f) {
    if(f.getNbHuman()){
        std::pair<std::shared_ptr<Humanoid>, bool> human = findHumanoidToAffect(f, Humanoid::Type::HUMAN);
        if (human.second) {
            _action = std::make_unique<KillAction>(human.first);
        } else {
            _action = std::make_unique<ChaseAction>(std::weak_ptr<Humanoid>(shared_from_this()), human.first);
        }
    } else {
        //Idle
    }

}

std::size_t Vampire::getMoveAmount() const {
    return 1;
}


```

File - KillerHumanoid.h

```h
#ifndef BUFFY_KILLERHUMANOID_H
#define BUFFY_KILLERHUMANOID_H

#include "../Humanoid.h"

/**
 * This class represents a humanoïd that can "kill" others (uses a KillAction).
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class KillerHumanoid : public Humanoid {

public:

    /**
     * Constructor for a new KillerHumanoid
     *
     * @param pos the pos of our new KillerHumanoid
     */
    explicit KillerHumanoid(const Point& pos);

    /**
     * Find a humanoid from a given type that is in range of our attack.
     *
     * @param f     the field of our simulation
     * @param type  the type of the humanoïd we want to kill
     * @return      the humanoïd of our given type that is the closest and returns true if the humanoid is in range, false otherwise.
     */
    std::pair<std::shared_ptr<Humanoid>, bool> findHumanoidToAffect(const Field& f, Humanoid::Type type) const;

    KillerHumanoid(size_t maxX, size_t maxY);

private:
    //The range of the KillerHumanoid
    size_t _range {1};
};

#endif  // BUFFY_KILLERHUMANOID_H

```

File - KillerHumanoid.cpp

```cpp
#include "KillerHumanoid.h"

KillerHumanoid::KillerHumanoid(size_t maxX, size_t maxY) : Humanoid(maxX, maxY) {}

KillerHumanoid::KillerHumanoid(const Point& pos) : Humanoid(pos) {}

std::pair<std::shared_ptr<Humanoid>, bool> KillerHumanoid::findHumanoidToAffect(const Field& f, Humanoid::Type type) const {
    // Get the current position of the Killer
    Point currPos = this->getPosition();

    //Get the closest type of humanoid
    std::shared_ptr<Humanoid> human = f.getClosest(std::const_pointer_cast<Humanoid>(shared_from_this()), type);

    //Return the closest humanoid and if it is in range
    return std::make_pair(human, human->getPosition().distance(currPos) <= _range);
}


```

File - Point.h

```cpp
#ifndef POINT_H
#define POINT_H
#include <cstddef>

/**
 * This class contains the data of a point in a 2D space.
 * It has some functions to calculate distance and direction to a point.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Point {
public:
    // x and y values in a 2D space
    size_t x;
    size_t y;

    /**
     * Constructor for a new Point.
     *
     * @param x x value of our Point
     * @param y y value of our Point
     */
    Point(size_t x, size_t y);

    /**
     * Gives us the distance between two given points.
     *
     * @param a first Point
     * @param b second Point
     * @return  the distance between Point a and Point b
     */
    static size_t distance(Point a, Point b);

    /**
     * Gives us the distance between this Point and another.
     *
     * @param other the other Point
     * @return      the distance between this and the other
     */
    size_t distance(Point other);

    /**
     * The direction we have to follow to ge to the other Point from this Point.
     *
     * @param other the Point where we want to go
     * @return      the direction to the other Point
     */
    [[nodiscard]] Point direction(Point other) const;
};


#endif  // POINT_H

```

File - Random.h

```cpp
#ifndef BUFFY_RANDOM_H
#define BUFFY_RANDOM_H

#include <random>
#include "../Field.h"
#include "Point.h"

/**
 * This class allow us to get random Point and random value between boundaries
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class Random {
private:
    // Static variable to get random numbers
    static std::random_device dev;
    static std::mt19937 rng;

public:
    /**
     * Give a random Point within the boundaries given ([min, max])
     *
     * @param minX  the minimum for the x value
     * @param maxX  the maximum for the x value
     * @param minY  the minimum for the y value
     * @param maxY  the maximum for the y value
     * @return      a new random Point
     */
    static Point getRandomPos(std::size_t minX, std::size_t maxX, std::size_t minY, std::size_t maxY);

    /**
     * Give a random size_t within the boundaries given ([min, max])
     *
     * @param min   minimum for size_t
     * @param max   maximum for size_t
     * @return      a random size_t
     */
    static std::size_t getRandom(std::size_t min, std::size_t max);
};

#endif  // BUFFY_RANDOM_H

```

File - Point.cpp

```cpp
#include "Point.h"
#include <cmath>
#include <cstddef>

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

size_t Point::distance(Point a, Point b) {
    int dx = std::abs(static_cast<int>(a.x) - static_cast<int>(b.x));
    int dy = std::abs(static_cast<int>(a.y) - static_cast<int>(b.y));
    return std::max(dx, dy);
}

size_t Point::distance(Point other) {
    return distance(*this, other);
}

Point Point::direction(Point other) const {
    size_t xDir = other.x - x;
    size_t yDir = other.y - y;
    return {xDir, yDir};
}

```

File - Random.cpp

```cpp
#include "Random.h"

std::mt19937 Random::rng(std::random_device{}());

Point Random::getRandomPos(std::size_t minX, std::size_t maxX, std::size_t minY, std::size_t maxY) {
    std::uniform_int_distribution<std::size_t> posX_dist{minX, maxX};
    std::uniform_int_distribution<std::size_t> posY_dist{minY, maxY};
    return Point{posX_dist(rng), posY_dist(rng)};
}
std::size_t Random::getRandom(std::size_t min, std::size_t max) {
    return std::uniform_int_distribution<std::size_t>{min, max}(rng);
}

```

File - BiteAction.h

```h
#ifndef BUFFY_BITEACTION_H
#define BUFFY_BITEACTION_H

#include "KillAction.h"

/**
 * This class derives from KillAction and allow us to maybe transform our target instead of killing it.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class BiteAction : public KillAction {
public:
    /**
     * Constructor for a new BiteAction.
     *
     * @param target    the humanoïd on which we apply the action
     */
    explicit BiteAction(const std::weak_ptr<Humanoid>& target);

    /**
     * Executes the action.
     * One chance out of 2 that it transforms the target into a Vampire or kills it.
     *
     * @param f the field of the simulation
     */
    void execute(Field &f) override;
};

#endif  // BUFFY_BITEACTION_H

```

File - KillAction.h

```cpp
#ifndef BUFFY_KILLACTION_H
#define BUFFY_KILLACTION_H

#include "../Action.h"

/**
 * This action kills the target it's given.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class KillAction : public Action {
public:
    /**
     * Constructor for a new KillAction.
     *
     * @param target    the humanoïd on which we apply the action
     */
    explicit KillAction(const std::weak_ptr<Humanoid>& target);

    /**
     * Executes the action.
     * Kills the target of this action.
     *
     * @param f the field of the simulation
     */
    void execute(Field &f) override;
};

#endif  // BUFFY_KILLACTION_H

```

File - MoveAction.h

```cpp
#include "../Action.h"
#include "../utilities/Point.h"

#ifndef BUFFY_MOVEACTION_H


/**
 * This class allow it's derived class to use the moveToPosition function.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class MoveAction : public Action {
public:
    /**
     * Constructor for a new MoveAction.
     *
     * @param target    the humanoïd on which we apply the action
     */
    explicit MoveAction(const std::weak_ptr<Humanoid>& target);

    /**
     * Move towards the position given.
     *
     * @param targetPos the position we want to move towards
     */
    void moveToPosition(const Point& targetPos);
};
#define BUFFY_MOVEACTION_H

#endif  // BUFFY_MOVEACTION_H

```

File - ChaseAction.h

```h
#ifndef BUFFY_CHASEACTION_H
#define BUFFY_CHASEACTION_H

#include "MoveAction.h"

/**
 * This action allow us to chase after a target of a given type.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class ChaseAction : public MoveAction {
public:
    /**
     * Constructor for a new ChaseAction.
     *
     * @param target    the humanoïd on which we apply the action
     * @param chasedTarget the target we're chasing
     */
    ChaseAction(const std::weak_ptr<Humanoid>& target,const std::weak_ptr<Humanoid>& chasedTarget);

    /**
     * Executes the action.
     * Moves towards the chased target.
     *
     * @param field
     */
    void execute(Field& field) override;

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

#endif  // BUFFY_CHASEACTION_H

```

File - BiteAction.cpp

```cpp
#include "BiteAction.h"
#include "../humanoids/Vampire.h"
#include "../utilities/Random.h"

BiteAction::BiteAction(const std::weak_ptr<Humanoid>& target) : KillAction(target){}

void BiteAction::execute(Field &f){
    const auto &human = _target.lock();
    if(!human) {
        return;
    }
    if (human->isAlive()) {
        if(Random::getRandom(0,1)){
            //Make a new vampire
            f.addHumanoid(std::make_shared<Vampire>(human->getPosition()));
            f.incrementVampireCount();
        }
        KillAction::execute(f);
    }
}


```

File - KillAction.cpp

```cpp
#include "KillAction.h"

KillAction::KillAction(const std::weak_ptr<Humanoid>& target) : Action(target) {

}

void KillAction::execute(Field &f){
    std::shared_ptr<Humanoid> t = _target.lock();
    if(t){
        t->dies();
    }
}

```

File - MoveAction.cpp

```cpp
#include "MoveAction.h"
#include <complex>
#include <iostream>

MoveAction::MoveAction(const std::weak_ptr<Humanoid>& target) : Action(target) {}

void MoveAction::moveToPosition(const Point& targetPos) {
    for(size_t i = 0; i < _target.lock()->getMoveAmount(); i++) {
        // Get the current position of the character
        Point currPos = _target.lock()->getPosition();

        int x_ = static_cast<int>(currPos.x - targetPos.x);
        int y_ = static_cast<int>(currPos.y - targetPos.y);

        Point newPos(currPos.x - (x_ == 0 ? 0 : x_ / std::abs(x_)) ,
                     currPos.y - (y_ == 0 ? 0 : y_ / std::abs(y_)));

        // Move the character to the new position
        _target.lock()->setPosition(newPos);
    }
}


```

File - ChaseAction.cpp

```cpp
#include "ChaseAction.h"

ChaseAction::ChaseAction(const std::weak_ptr<Humanoid>& target,const std::weak_ptr<Humanoid>& chasedTarget) : MoveAction(target), _chasedTarget(chasedTarget) {}

void ChaseAction::execute(Field& field) {
    std::shared_ptr<Humanoid> hum = _chasedTarget.lock();
    if (hum) {
        moveToPosition(hum->getPosition());
        return;
    }
}

```

File - MoveRandomAction.h

```h
#ifndef BUFFY_MOVERANDOMACTION_H
#define BUFFY_MOVERANDOMACTION_H

#include "MoveAction.h"

/**
 * This action picks a random position and move towards it.
 *
 * @author Roland Samuel
 * @author Haeffner Edwin
 * @author Junod Arthur
 */
class MoveRandomAction : public MoveAction {
public:
    /**
     * Constructor for a new MoveRandomAction.
     *
     * @param target    the humanoïd on which we apply the action
     */
    explicit MoveRandomAction(const std::weak_ptr<Humanoid>& target);

    /**
     * Executes the action.
     * Get a random position and move towards it.
     *
     * @param f the field fo the simluation
     */
    void execute(Field& f) override;
};

#endif  // BUFFY_MOVERANDOMACTION_H

```

File - MoveRandomAction.cpp

```cpp
#include "MoveRandomAction.h"
#include "../utilities/Random.h"

MoveRandomAction::MoveRandomAction(const std::weak_ptr<Humanoid>& target) : MoveAction(target) {}

void MoveRandomAction::execute(Field& f) {
    moveToPosition(Random::getRandomPos(0, f.getWidth() - 1, 0, f.getHeight() - 1));
}

```