```
#ifndef BUFFY_FIELD_HPP
#define BUFFY_FIELD_HPP

#include <list>
#include "actors/Humanoid.hpp"
#include "actors/Buffy.hpp"
#include "actors/Human.hpp"
#include "actors/Vampire.hpp"
#include "EndStatus.hpp"

class Controller;

/**
 * Classe Field représentant le terrain de la simulation sur lequel évolue les
 * acteurs
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Field {
public:
    /**
     * Constructeur de la classe Field
     * @param width : largeur du Field
     * @param height : hauteur du Field
     * @param nbHumans : nombre d'humains dans le Field
     * @param nbVampires : nombre de vampires dans le Field
     */
    Field(unsigned width, unsigned height,
            unsigned nbHumans, unsigned nbVampires);

    /**
     * Destructeur de la classe Field supprimant les allocations dynamiques
     */
    ~Field();

    /**
     * Empêche la copie d'un Field
     */
    Field(const Field&) = delete;

    /**
     * Empêche re-affectation d'un Field
     */
    Field& operator=(const Field&) = delete;

    /**
     * Augmente d'un tour le nombre de tours passés dans la simulation
     * @return le numéro du tour actuel
     */
    int nextTurn();

    /**
     * Méthode permettant de trouver l'humanoïde le plus proche d'un autre
     * @tparam T : type de l'humanoïde à rechercher
     * @param closeTo : humanoïde à comparer
     * @return l'humanoïde le plus proche du type souhaité (nullptr si aucun)
     */
    template<typename T>
    T* findClosestHumanoid(const Humanoid& closeTo) const;

    /**
     * Ajoute un humanoïde à la liste des humanoïdes en jeu
     *
     * Remarque: si l'humanoïde est déjà présent, ne fait rien
     * @param humanoid : humanoïde à ajouter
     */
    void addCharacter(Humanoid* humanoid);

    /**
     * @return true s'il reste des humains vivants, false sinon
     */
    bool hasHumans() const;
```

```
    /**
     * @return true s'il reste des vampires vivants, false sinon
     */
    bool hasVampires() const;

    /**
     * Indique à la simulation qu'un humain est mort
     */
    void humanDied();

    /**
     * Indique à la simulation qu'un vampire est mort
     */
    void vampireDied();

    /**
     * Indique à la simulation qu'un nouveau vampire est né
     */
    void vampireBorn();

    /**
     * @return la largeur du Field
     */
    unsigned getWidth() const;

    /**
     * @return la hauteur du Field
     */
    unsigned getHeight() const;

    /**
     * @return un itérateur constant sur le premier acteur du Field
     */
    std::list<Humanoid*>::const_iterator begin() const;

    /**
     * @return un itérateur constant après le dernier acteur du Field
     */
    std::list<Humanoid*>::const_iterator end() const;

    /**
     * @return un status concernant la fin potentielle de la simulation
     */
    EndStatus isFinished() const;

private:
    unsigned width, height;
    int turn;
    unsigned nbHumans, nbVampires;
    std::list<Humanoid*> humanoids;
};

#include "Field_Impl.hpp"

#endif // BUFFY_FIELD_HPP


/**
 * Implémentation des méthodes génériques de Field
 *
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#ifndef BUFFY_FIELD_IMPL_HPP
#define BUFFY_FIELD_IMPL_HPP

#include <limits>

template<typename T>
T* Field::findClosestHumanoid(const Humanoid& closeTo) const {
```

```
    double minDist = std::numeric_limits<double>::max();
    T* closest = nullptr;
    T* converted;

    for (Humanoid* humanoid: humanoids) {
        if ((converted = dynamic_cast<T*>(humanoid)) == nullptr)
            continue;

        double dist = humanoid->getPosition().getDistance(closeTo.getPosition());
        if (dist < minDist) {
            minDist = dist;
            closest = converted;
        }
    }
    return closest;
}


#endif // BUFFY_FIELD_IMPL_HPP

/**
 * Classe Field représentant le terrain de la simulation sur lequel évolue les
 * acteurs
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Field.hpp"
#include "utils/Random.hpp"

using namespace std;

Field::Field(unsigned width, unsigned height, unsigned nbHumans, unsigned nbVampires)
    : width(width), height(height), turn(0),
      nbHumans(nbHumans), nbVampires(nbVampires), humanoids() {

    int maxX = (int)width;
    int maxY = (int)height;

    for (unsigned i = 0; i < nbVampires; ++i)
        humanoids.emplace_back(new Vampire(Position::getRandomPosition(maxX, maxY)));
    for (unsigned i = 0; i < nbHumans; ++i)
        humanoids.emplace_back(new Human(Position::getRandomPosition(maxX, maxY)));

    humanoids.emplace_back(new Buffy(Position::getRandomPosition(maxX, maxY)));
}

Field::~Field() {
    for (Humanoid* humanoid: humanoids)
        delete humanoid;
}

int Field::nextTurn() {
    // Déterminer les prochaines actions
    for (auto& humanoid: humanoids)
        humanoid->setAction(*this);

    // Exécuter les actions
    for (auto& humanoid: humanoids)
        humanoid->executeAction(*this);

    // Enlever les humanoides tués
    for (auto it = humanoids.begin(); it != humanoids.end();) {
        if (!(*it)->isAlive()) {
            delete *it;
            it = humanoids.erase(it);
        } else
            ++it;
    }
    return turn++;
}
```

```
void Field::addCharacter(Humanoid* humanoid) {
    auto end = this->end();
    for (auto i = this->begin(); i != end; ++i) {
        if (*i == humanoid)
            return;
    }
    humanoids.emplace_back(humanoid);
}

bool Field::hasHumans() const {
    return nbHumans > 0;
}

bool Field::hasVampires() const {
    return nbVampires > 0;
}

void Field::humanDied() {
    if (nbHumans)
        --nbHumans;
}

void Field::vampireDied() {
    if (nbVampires)
        --nbVampires;
}

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

unsigned Field::getWidth() const {
    return width;
}

unsigned Field::getHeight() const {
    return height;
}

list<Humanoid*>::const_iterator Field::begin() const {
    return humanoids.begin();
}

list<Humanoid*>::const_iterator Field::end() const {
    return humanoids.end();
}

EndStatus Field::isFinished() const {
    if (nbHumans > 0)
        return nbVampires > 0 ? EndStatus::RUNNING : EndStatus::WIN;
    return EndStatus::LOSE;
}


#ifndef BUFFY_CONTROLLER_HPP
#define BUFFY_CONTROLLER_HPP

#include "Field.hpp"

class Displayer;

/**
 * Classe Controller gérant toute la logique globale du programme
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Controller {
public:
    /**
     * Crée un nouveau contrôleur
     * @param width : largeur du Field
     * @param height : hauteur du Field
```

```
     * @param nbHumans : nombre d'humains initial
     * @param nbVampires : nombre de vampires initial
     * @param displayer : afficheur
     * @throws invalid_argument si des paramètres valent 0
     */
    Controller(unsigned width, unsigned height,
                  unsigned nbHumans, unsigned nbVampires,
                  Displayer& displayer);

    /**
     * Lance la simulation
     */
    void run();

private:
    void nextTurn();

    void quit();

    void statistics();

    void handleCommand();

    Displayer* displayer;
    Field field;
    int turn;
    bool finished;

    unsigned width, height;
    unsigned nbHumans, nbVampires;

    static constexpr unsigned NB_SIMULATIONS = 10000;

    static constexpr char
        QUIT = 'q',
        STATS = 's',
        NEXT = 'n';
};

#endif // BUFFY_CONTROLLER_HPP


/**
 * Classe Controller gérant toute la logique globale du programme
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Controller.hpp"
#include "displayers/Displayer.hpp"
#include "Field.hpp"
#include "EndStatus.hpp"
#include <iostream>

using namespace std;

Controller::Controller(unsigned width, unsigned height,
                               unsigned nbHumans, unsigned nbVampires,
                               Displayer& displayer)
    : displayer(&displayer), field(width, height, nbHumans, nbVampires),
      turn(0), finished(false), width(width), height(height),
      nbHumans(nbHumans), nbVampires(nbVampires) {

    if (width == 0 || height == 0)
        throw invalid_argument("Erreur: Les dimensions du Field doivent etre > 0.");

    if (nbHumans == 0 || nbVampires == 0)
        throw invalid_argument(
            "Erreur: La simulation demande au moins un humain et un vampire."
        );
}
```

```
void Controller::run() {
    turn = 0;
    displayer->display(field);

    while (!finished) {
        handleCommand();
    }
}

void Controller::nextTurn() {
    turn = field.nextTurn();
    displayer->display(field);
}

void Controller::handleCommand() {
    do {
        Displayer::displayPrompt(turn, QUIT, STATS, NEXT);

        string line;
        getline(cin, line);
        char input = (line.empty() ? NEXT : line[0]);

        switch ((char)tolower(input)) {
            case QUIT:
                quit();
                return;
            case STATS:
                statistics();
                return;
            case NEXT:
                nextTurn();
                return;
            default:
                break;
        }
    } while (true);
}

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

void Controller::statistics() {
    displayer->displayStarting();
    unsigned wins = 0;
    for (unsigned i = 0; i < NB_SIMULATIONS; ++i) {
        Field simulation(width, height, nbHumans, nbVampires);

        while (simulation.isFinished() == EndStatus::RUNNING) {
            simulation.nextTurn();
        }

        if (simulation.isFinished() == EndStatus::WIN)
            ++wins;
    }
    displayer->displayStats(wins * 100.0 / NB_SIMULATIONS, NB_SIMULATIONS);
}


#ifndef BUFFY_ENDSTATUS_HPP
#define BUFFY_ENDSTATUS_HPP

/**
 * Enum représentant les différents statuts possibles du jeu au cours de la
 * simulation
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
enum class EndStatus {
    RUNNING,
    WIN,
    LOSE
```

```
};

#endif // BUFFY_ENDSTATUS_HPP


#ifndef BUFFY_HUMANOID_HPP
#define BUFFY_HUMANOID_HPP

#include "../actions/Action.hpp"
#include "../utils/Position.hpp"
#include "../displayers/Color.hpp"

class Field;

/**
 * Classe Humanoid représentant tous les acteurs de la simulation
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Humanoid {
public:
    /**
     * Crée un nouvel humanoïde à une donnée
     * @param position : position de l'humanoïde
     */
    explicit Humanoid(const Position& position);

    /**
     * Empêche la copie d'un humanoïde
     */
    Humanoid(const Humanoid&) = delete;

    /**
     * Empêche re-affectation d'un humanoïde
     */
    Humanoid& operator=(const Humanoid&) = delete;

    /**
     * Destructeur supprimant l'allocation dynamique de l'action
     */
    virtual ~Humanoid();

    /**
     * Assigne une action à effectuer lors du prochain tour
     * @param field : Field sur lequel l'action doit être effectuée
     */
    void setAction(const Field& field);

    /**
     * Execute l'action associée à l'humanoïde
     * @param field
     */
    void executeAction(Field& field);

    /**
     * @return true si l'humanoïde est vivant, false sinon
     */
    virtual bool isAlive() const;

    /**
     * Tue l'humanoïde
     * @param field : field sur lequel l'humanoïde se trouve
     */
    virtual void kill(Field& field);

    /**
     * @return la position courante de l'humanoïde
     */
    const Position& getPosition() const;

    /**
     * Définit la nouvelle position de l'humanoïde
```

```
     * @param position : nouvelle position de l'humanoïde
     */
    void setPosition(const Position& position);

    /**
     * @return le symbole associé à l'humanoïde
     */
    virtual char getSymbol() const = 0;

    /**
     * @return la couleur associée à l'humanoïde
     */
    virtual Color getColor() const = 0;

protected:
    /**
     * @param field : field sur lequel l'action doit être effectuée
     * @return la prochaine action à effectuer
     */
    virtual Action* getNextAction(const Field& field) = 0;

private:
    bool alive;
    Position position;
    Action* action;
};

#endif // BUFFY_HUMANOID_HPP


/**
 * Classe Humanoid représentant tous les acteurs de la simulation
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

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

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

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

void Humanoid::setAction(const Field& field) {
    action = getNextAction(field);
}

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

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

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

const Position& Humanoid::getPosition() const {
```

```
    return position;
}

void Humanoid::setPosition(const Position& _position) {
    position = _position;
}


#ifndef BUFFY_BUFFY_HPP
#define BUFFY_BUFFY_HPP

#include "Humanoid.hpp"

/**
 * Classe Buffy représentant un super-vampire qui peut tuer les autres vampires
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Buffy : public Humanoid {
public:
    /**
     * Crée une nouvelle Buffy à une position donnée
     * @param position : position de Buffy
     */
    explicit Buffy(const Position& position);

    char getSymbol() const override;

    Color getColor() const override;

protected:
    Action* getNextAction(const Field& field) override;

private:
    static constexpr int
        HUNT_RANGE = 1,
        MOVE_RANGE = 2;
};

#endif // BUFFY_BUFFY_HPP


/**
 * Classe Buffy représentant un super-vampire qui peut tuer les autres vampires
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Buffy.hpp"
#include "Human.hpp"
#include "Vampire.hpp"
#include "../Field.hpp"
#include "../actions/Kill.hpp"
#include "../actions/Move.hpp"

using namespace std;

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

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

Color Buffy::getColor() const {
    return Color::YELLOW;
}

Action* Buffy::getNextAction(const Field& field) {
    if (!field.hasVampires())
        return new Move(Human::MOVE_RANGE, *this);

    Vampire* target = field.findClosestHumanoid<Vampire>(*this);
```

```
    if (getPosition().getDistance(target->getPosition()) <= HUNT_RANGE)
        return new Kill(*target);

    return new Move(MOVE_RANGE, *this, target);
}


#ifndef BUFFY_HUMAN_HPP
#define BUFFY_HUMAN_HPP

#include "Humanoid.hpp"

/**
 * Classe Humain représentant un acteur se déplaçant aléatoirement dans la
 * simulation et pouvant être transformé en vampire
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Human : public Humanoid {
public:
    /**
     * Crée un humain à une position donnée
     * @param position : position de l'humain
     */
    explicit Human(const Position& position);

    void kill(Field& field) override;

    char getSymbol() const override;

    Color getColor() const override;

    /**
     * Portée de déplacement des humains
     */
    static constexpr int MOVE_RANGE = 1;

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

#endif // BUFFY_HUMAN_HPP


/**
 * Classe Humain représentant un acteur se déplaçant aléatoirement dans la
 * simulation et pouvant être transformé en vampire
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Human.hpp"
#include "../actions/Move.hpp"
#include "../Field.hpp"

using namespace std;

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

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

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

Color Human::getColor() const {
    return Color::PINK;
}
```

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


#ifndef BUFFY_VAMPIRE_HPP
#define BUFFY_VAMPIRE_HPP

#include "Humanoid.hpp"

/**
 * Classe Vampire représentant un acteur chassant les humains et pouvant transformer
 * les transformer en vampires
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Vampire : public Humanoid {
public:
    /**
     * Crée un nouveau Vampire à un position donnée
     * @param position : position du vampire
     */
    explicit Vampire(const Position& position);

    void kill(Field& field) override;

    char getSymbol() const override;

    Color getColor() const override;

protected:
    Action* getNextAction(const Field& field) override;

private:
    static constexpr int
        HUNT_RANGE = 1,
        MOVE_RANGE = 1;
};

#endif // BUFFY_VAMPIRE_HPP


/**
 * Classe Vampire représentant un acteur chassant les humains et pouvant transformer
 * les transformer en vampires
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Vampire.hpp"
#include "Human.hpp"
#include "../Field.hpp"
#include "../actions/Kill.hpp"
#include "../actions/Move.hpp"
#include "../actions/Transform.hpp"
#include "../utils/Random.hpp"

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

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

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

Color Vampire::getColor() const {
    return Color::BLUE;
}
```

```
Action* Vampire::getNextAction(const Field& field) {
    if (!field.hasHumans())
        return nullptr;

    Human* target = field.findClosestHumanoid<Human>(*this);
    if (getPosition().getDistance(target->getPosition()) <= HUNT_RANGE) {
        // 50% de chance de tuer, 50% de chance de transformer
        if (Random::generateBool())
            return new Kill(*target);
        else
            return new Transform(*target);
    }

    return new Move(MOVE_RANGE, *this, target);
}
```

```
#ifndef BUFFY_ACTION_HPP
#define BUFFY_ACTION_HPP

class Field;

class Humanoid;

/**
 * Classe abstraite représentant une action réalisée par un humanoïde de la
 * simulation
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Action {
public:
    /**
     * Crée une nouvelle action
     * @param humanoid : humanoïde qui effectue/subit l'action
     */
    explicit Action(Humanoid& humanoid);

    /**
     * Destructeur par défaut de la classe Action
     */
    virtual ~Action() = default;

    /**
     * Empêche la copie d'une Action
     */
    Action(const Action&) = delete;

    /**
     * Empêche re-affectation d'une Action
     */
    Action& operator=(const Action&) = delete;

    /**
     * Execute l'action
     * @param field : Field sur lequel l'action doit être effectuée
     */
    virtual void execute(Field& field) = 0;

protected:
    /**
     * @return l'humanoïde associé à l'action
     */
    Humanoid* getHumanoid();

private:
    Humanoid* humanoid;
};

#endif // BUFFY_ACTION_HPP
```

```
/**
 * Classe abstraite représentant une action réalisée par un humanoïde de la
 * simulation
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

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

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

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


#ifndef BUFFY_KILL_HPP
#define BUFFY_KILL_HPP

#include "Action.hpp"

/**
 * Classe Kill représentant la mort de l'humanoïde souhaité
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Kill : public Action {
public:
    /**
     * Crée une nouvelle action d'homicide
     * @param humanoid : humanoïde à tuer
     */
    explicit Kill(Humanoid& humanoid);

    void execute(Field& field) override;
};

#endif // BUFFY_KILL_HPP


/**
 * Classe Kill représentant la mort de l'humanoïde souhaité
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

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

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

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


#ifndef BUFFY_TRANSFORM_HPP
#define BUFFY_TRANSFORM_HPP

#include "Action.hpp"

class Humanoid;

/**
 * Classe Transform représentant la transformation d'un humain en vampire
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
```

```
 */
class Transform : public Action {
public:
    /**
     * Crée une nouvelle action de transformation
     * @param humanoid : humanoïde à transformer
     */
    explicit Transform(Humanoid& humanoid);

    void execute(Field& field) override;
};

#endif // BUFFY_TRANSFORM_HPP


/**
 * Classe Transform représentant la transformation d'un humain en vampire
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Transform.hpp"
#include "../actors/Humanoid.hpp"
#include "../actors/Vampire.hpp"
#include "../Field.hpp"

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

void Transform::execute(Field& field) {
    if (getHumanoid()->isAlive()) {
        getHumanoid()->kill(field);
        field.addCharacter(new Vampire(getHumanoid()->getPosition()));
        field.vampireBorn();
    }
}


#ifndef BUFFY_MOVE_HPP
#define BUFFY_MOVE_HPP

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

class Humanoid;

/**
 * Classe Move représentant le déplacement d'un humanoïde
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Move : public Action {
public:
    /**
     * Crée un nouveau déplacement (aléatoire si aucun cible n'est spécifiée)
     * @param range : longueur du déplacement
     * @param humanoid : Humanoid à déplacer
     * @param target : potentielle cible à atteindre
     */
    Move(unsigned range, Humanoid& humanoid, const Humanoid* target = nullptr);

    void execute(Field& f) override;

private:
    std::vector<const Position*> getPossibleDirections(const Position& position,
                                                       const Field& field) const;

    unsigned range;
    const Humanoid* target;
};
```

```
#endif // BUFFY_MOVE_HPP


/**
 * Classe Move représentant le déplacement d'un humanoïde
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Move.hpp"
#include "../utils/Random.hpp"
#include "../actors/Humanoid.hpp"
#include "../Field.hpp"

using namespace std;

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

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

    for (unsigned i = 0; i < range; ++i) {
        if (target) {
            direction = newPosition.getDirection(target->getPosition());
        } else {
            vector<const Position*> directions =
                getPossibleDirections(newPosition, field);

            if (directions.empty())
                break;

            direction = *directions.at(
                (unsigned long)(Random::generate((int)directions.size()))
            );
        }
        newPosition.add(direction);
    }

    getHumanoid()->setPosition(newPosition);
}

vector<const Position*> Move::getPossibleDirections(const Position& position,
                                                    const Field& field) const {
    vector<const Position*> possibleDirections;

    int x = position.getX();
    int y = position.getY();
    int maxX = (int)field.getWidth();
    int maxY = (int)field.getHeight();
    int reach = (int)range;

    if (x >= reach)
        possibleDirections.push_back(&Position::LEFT);
    if (x >= reach && y >= reach)
        possibleDirections.push_back(&Position::UP_LEFT);
    if (x >= reach && y < maxY - reach)
        possibleDirections.push_back(&Position::DOWN_LEFT);
    if (x < maxX - reach)
        possibleDirections.push_back(&Position::RIGHT);
    if (x < maxX - reach && y >= reach)
        possibleDirections.push_back(&Position::UP_RIGHT);
    if (x < maxX - reach && y < maxY - reach)
        possibleDirections.push_back(&Position::DOWN_RIGHT);
    if (y >= reach)
        possibleDirections.push_back(&Position::UP);
    if (y < maxY - reach)
        possibleDirections.push_back(&Position::DOWN);

    return possibleDirections;
```

```
}


#ifndef BUFFY_POSITION_HPP
#define BUFFY_POSITION_HPP

/**
 * Classe Position représentant une position dans la simulation en 2 dimensions
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Position {
public:
    /**
     * Crée une nouvelle position aux coordonnées (0, 0)
     */
    Position();

    /**
     * Crée une position aux coordonnées souhaitées
     * @param x : coordonnée x
     * @param y : coordonnée y
     */
    Position(int x, int y);

    /**
     * @return la coordonnée x
     */
    int getX() const;

    /**
     * @return la coordonnée y
     */
    int getY() const;

    /**
     * Ajoute une Position à celle-ci
     * @param other : position à ajouter
     * @return position initiale modifiée
     */
    Position& add(const Position& other);

    /**
     * Récupérer la direction permettant d'aller à la position souhaitée
     * @param to : position d'arrivée souhaitée
     * @return la direction
     */
    Position getDirection(const Position& to) const;

    /**
     * Calcule la distance jusqu'à la position souhaitée
     * @param to : position d'arrivée souhaitée
     * @return la distance
     */
    int getDistance(const Position& to) const;

    /**
     * Génère une position aléatoire
     * @param maxX : borne max de la position x
     * @param maxY : borne max de la position y
     * @return la position aléatoire créée
     */
    static Position getRandomPosition(int maxX, int maxY);

    /**
     * Différentes directions possibles
     */
    static const Position
        UP,
        UP_LEFT,
        LEFT,
        DOWN_LEFT,
```

```
        DOWN,
        DOWN_RIGHT,
        RIGHT,
        UP_RIGHT;

private:
    int x;
    int y;
};

#endif // BUFFY_POSITION_HPP


/**
 * Classe Position représentant une position dans la simulation en 2 dimensions
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

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

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

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

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

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

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

Position& Position::add(const Position& other) {
    x += other.x;
    y += other.y;
    return *this;
}

Position Position::getDirection(const Position& to) const {
    int _x = to.x - x;
    int _y = to.y - y;

    return {
        _x == 0 ? 0 : _x / abs(_x),
        _y == 0 ? 0 : _y / abs(_y)
    };
}

int Position::getDistance(const Position& to) const {
    double first = abs((x - to.x));
    double second = abs((y - to.y));
    return (int)round(hypot(first, second));
}

Position Position::getRandomPosition(int maxX, int maxY) {
    return {
        Random::generate(maxX),
        Random::generate(maxY)
    };
```

```
}


#ifndef BUFFY_RANDOM_HPP
#define BUFFY_RANDOM_HPP

#include <random>

/**
 * Classe permettant de générer facilement des nombres aléatoires
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Random {
public:
    /**
     * Génère un nombre aléatoire entre 0 et max non compris
     * @param max : borne supérieure
     * @return le nombre aléatoire
     * @throws invalid_argument si max <= 0
     */
    static int generate(int max);

    /**
     * Génère un nombre aléatoire dans l'intervalle [min, max[
     * @param min : borne inférieure
     * @param max : borne supérieure
     * @return le nombre aléatoire
     * @throws invalid_argument si max <= min
     */
    static int generate(int min, int max);

    /**
     * Génère un boolean true ou false
     * @return le boolean aléatoire
     */
    static bool generateBool();

private:
    static std::mt19937 generator;
};

#endif // BUFFY_RANDOM_HPP


/**
 * Classe permettant de générer facilement des nombres aléatoires
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

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

using namespace std;

mt19937 Random::generator(
    (unsigned)chrono::system_clock::now().time_since_epoch().count()
);

int Random::generate(int min, int max) {
    if (max <= min)
        throw invalid_argument("La valeur min doit etre plus grande que max.");

    uniform_int_distribution<int> distribution(min, max - 1);
    return distribution(generator);
}

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

```
bool Random::generateBool() {
    int random = generate(0, 2);
    return random == 1;
}


#ifndef BUFFY_COLOR_HPP
#define BUFFY_COLOR_HPP

/**
 * Enum représentant les différentes couleurs possibles dans l'affichage
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
enum class Color {
    BLUE,
    YELLOW,
    PINK,
};

#endif // BUFFY_COLOR_HPP


#ifndef BUFFY_DISPLAYER_HPP
#define BUFFY_DISPLAYER_HPP

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

class Field;

/**
 * Classe Displayer permettant d'afficher la simulation dans la console
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class Displayer {
public:
    /**
     * Crée un Displayer
     * @param width : largeur du Field à afficher
     * @param height : hauteur du Field à afficher
     */
    Displayer(unsigned width, unsigned height);

    /**
     * Affiche le Field ainsi que son contenu
     * @param field : field à afficher
     */
    virtual void display(const Field& field);

    /**
     * Affiche l'humanoïde
     * @param humanoid : humanoïde à afficher
     */
    virtual void display(const Humanoid* humanoid) const;

    /**
     * Affiche une message indiquant que le calcul des statistiques est en cours
     */
    virtual void displayStarting() const;

    /**
     * Affiche le résultat des statistiques
     * @param winrate : pourcentage de victoire
     * @param total : nombre total de parties
     */
    virtual void displayStats(double winrate, unsigned total) const;

    /**
```

```
     * Afficher le menu du programme
     * @param turn : tour actuel
     * @param quit : caractère représentant l'option quitter
     * @param stats : caractère représentant l'option statistiques
     * @param next : caractère représentant l'option tour suivant
     */
    static void displayPrompt(int turn, char quit, char stats, char next);

private:
    static void displayHorizontalBorder(const Field& field);

    static constexpr char
        CORNER = '+',
        HORIZONTAL_BORDER = '-',
        VERTICAL_BORDER = '|',
        EMPTY = ' ';

    std::vector<std::vector<const Humanoid*>> content;
};

#endif // BUFFY_DISPLAYER_HPP


/**
 * Classe Displayer permettant d'afficher la simulation dans la console
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "Displayer.hpp"
#include "../Field.hpp"
#include <iostream>
#include <iomanip>

using namespace std;

Displayer::Displayer(unsigned width, unsigned height)
    : content(height, vector<const Humanoid*>(width, nullptr)) {
}

void Displayer::display(const Field& field) {
    Position position;
    const Humanoid** toDisplay;

    // Ajoute les acteurs à leur position
    for (auto humanoid: field) {
        position = humanoid->getPosition();
        content.at((unsigned)position.getY()).at((unsigned)position.getX())
            = humanoid;
    }

    displayHorizontalBorder(field);
    for (unsigned y = 0; y < field.getHeight(); y++) {
        cout << VERTICAL_BORDER;
        for (unsigned x = 0; x < field.getWidth(); x++) {
            toDisplay = &content.at(y).at(x);
            if (*toDisplay) {
                display(*toDisplay);
                *toDisplay = nullptr;
            } else
                cout << EMPTY;
        }
        cout << VERTICAL_BORDER << endl;
    }
    displayHorizontalBorder(field);
}

void Displayer::display(const Humanoid* humanoid) const {
    cout << humanoid->getSymbol();
}

void Displayer::displayStarting() const {
```

```
    cout << "Statistics are beeing calculated..." << endl;
}

void Displayer::displayStats(double winrate, unsigned total) const {
    cout << "\rBuffy's win rate: " << left << setw(6) << setprecision(2) << fixed <<
          winrate << "% " << "(" << total << " iterations)" << endl;
}

void Displayer::displayPrompt(int turn, char quit, char stats, char next) {
    cout << "[" << turn << "] "
          << quit << ")uit "
          << stats << ")tatistics "
          << next << ")ext: ";
}

void Displayer::displayHorizontalBorder(const Field& field) {
    cout << CORNER << setfill(HORIZONTAL_BORDER)
          << setw((int)field.getWidth() + 1) << CORNER << endl;

    cout << setfill(EMPTY);
}


#ifndef BUFFY_WINDOWSDISPLAYER_HPP
#define BUFFY_WINDOWSDISPLAYER_HPP

#ifdef __WIN32

#include "Displayer.hpp"
#include <windows.h>

/**
 * Classe Windows permettant d'afficher la simulation dans la console avec
 * des couleurs sur Windows
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class WindowsDisplayer : public Displayer {
public:
    /**
     * Crée un WindowsDisplayer spécifique à Windows
     * @param width : largeur du Field à afficher
     * @param height : hauteur du Field à afficher
     */
    WindowsDisplayer(unsigned width, unsigned height);

    void display(const Humanoid* humanoid) const override;

private:
    static WORD getCurrentColor();

    static void changeColor(WORD color);

    WORD getColor(Color color) const;

    WORD defaultColor;
};

#endif // __WIN32

#endif // BUFFY_WINDOWSDISPLAYER_HPP



/**
 * Classe Windows permettant d'afficher la simulation dans la console avec
 * des couleurs sur Windows
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include "WindowsDisplayer.hpp"
```

```
#ifdef __WIN32

using namespace std;

WindowsDisplayer::WindowsDisplayer(unsigned width, unsigned height)
    : Displayer(width, height), defaultColor(getCurrentColor()) {
}

void WindowsDisplayer::display(const Humanoid* humanoid) const {
    changeColor(getColor(humanoid->getColor()));
    Displayer::display(humanoid);
    changeColor(defaultColor);
}

WORD WindowsDisplayer::getColor(Color color) const {
    switch (color) {
        case Color::BLUE:
            return 0x0D;
        case Color::YELLOW:
            return 0x0E;
        case Color::PINK:
            return 0x01;
        default:
            return defaultColor;
    }
}

WORD WindowsDisplayer::getCurrentColor() {
    CONSOLE_SCREEN_BUFFER_INFO info;
    GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &info);
    return info.wAttributes;
}

void WindowsDisplayer::changeColor(WORD color) {
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), color);
}

#endif // __WIN32



#ifndef BUFFY_UNIXDISPLAYER_HPP
#define BUFFY_UNIXDISPLAYER_HPP

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

/**
 * Classe UnixDisplayer permettant d'afficher la simulation dans la console avec
 * des couleurs sur les plateformes Unix
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */
class UnixDisplayer : public Displayer {
public:
    /**
     * Crée un UnixDisplayer spécifique aux plateformes Unix
     * @param width : largeur du Field à afficher
     * @param height : hauteur du Field à afficher
     */
    UnixDisplayer(unsigned width, unsigned height);

    void display(const Humanoid* humanoid) const override;

private:
    static std::string getColor(Color color);

    static void resetColor();
};

#endif // BUFFY_UNIXDISPLAYER_HPP
```

```
/**
 * Classe UnixDisplayer permettant d'afficher la simulation dans la console avec
 * des couleurs sur les plateformes Unix
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

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

using namespace std;

UnixDisplayer::UnixDisplayer(unsigned width, unsigned height)
    : Displayer(width, height) {
}

void UnixDisplayer::display(const Humanoid* humanoid) const {
    cout << getColor(humanoid->getColor());
    Displayer::display(humanoid);
    resetColor();
}

string UnixDisplayer::getColor(Color color) {
    switch (color) {
        case Color::BLUE:
            return "\033[1m\033[34m";
        case Color::YELLOW:
            return "\033[1m\033[33m";
        case Color::PINK:
            return "\033[1m\033[35m";
        default:
            return "";
    }
}

void UnixDisplayer::resetColor() {
    cout << "\033[0m";
}
```

```
/**
 * Point d'entrée du programme.
 * @author Alexandre Jaquier
 * @author Valentin Kaelin
 */

#include <iostream>
#include "Controller.hpp"
#include "displayers/Displayer.hpp"
#include "displayers/UnixDisplayer.hpp"
#include "displayers/WindowsDisplayer.hpp"

using namespace std;

/**
 * Point d'entrée du programme
 * @param argc : nombre d'arguments
 * @param argv : tableau des arguments
 * @return le code de sortie du programme
 */
int main(int argc, char* argv[]) {
    const int NB_ARGS = 5;

    if (argc != NB_ARGS)
        cout << "Usage: " << argv[0]
             << " <width> <height> <nbHumans> <nbVampires>" << endl;

    unsigned width, height, nbHumans, nbVampires;
```

```
    try {
        for (int i = 1; i < argc; ++i) {
            if (stoi(argv[i]) < 0) {
                // Vérification que les différentes tailles ne sont pas négatives
                throw exception();
            }
        }
        width = (unsigned)stoi(argv[1]);
        height = (unsigned)stoi(argv[2]);
        nbHumans = (unsigned)stoi(argv[3]);
        nbVampires = (unsigned)stoi(argv[4]);
    } catch (exception& e) {
        throw invalid_argument("Erreur: Un argument du programme est invalide.");
    }

#if  defined(__linux__) || defined(__APPLE__)
    UnixDisplayer displayer(width, height);
#elif __WIN32
    WindowsDisplayer displayer(width, height);
#else
    Displayer displayer(width, height);
#endif

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

    return EXIT_SUCCESS;
}
```