```cpp
/**
 * Classe représentant le point d'entrée du programme
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include <iostream>
#include "simulation/Simulator.hpp"
#include "displayers/ConsoleDisplayer.hpp"

#define NB_ARGS 4
#define NB_SIMULATION 10000

int main(int argc, char* argv[]) {
   unsigned values[NB_ARGS];

   if (argc != NB_ARGS + 1) {
      throw std::invalid_argument("Usage : buffy <largeur grille> <hauteur grille> <nb vampires>"
                                  " <nb humains>");
   }
   try{
       for (unsigned i = 1; i < NB_ARGS + 1; ++i) {
           if (std::stoi(argv[i]) <= 0) {
               throw std::exception();
           }
           values[i - 1] = (unsigned)std::stoi(argv[i]);
       }
   }catch (const std::exception&){ // Permet de catch les exceptions de std::stoi
        // et si la valeur est < 1
       throw std::invalid_argument("Les arguments doivent etre des nombres entiers positifs !");
   }


   unsigned width = values[0], height = values[1], nbVampires = values[2], nbHumans = values[3];
   ConsoleDisplayer displayer(width, height);

   Simulator simulator(NB_SIMULATION, width, height, nbVampires, nbHumans, displayer);
   simulator.run();

   return EXIT_SUCCESS;
}

/* ---------------------------------------------------------------------------------------------*/


#ifndef BUFFY_SIMULATOR_HPP
#define BUFFY_SIMULATOR_HPP

#include <string>
#include "../displayers/Displayer.hpp"
#include "Field.hpp"
#include "utils/Random.hpp"

/**
 * Classe contrôlant la simulation du jeu
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Simulator {
public:
```

```
70      /**
71       * Constructeur de base
72       * @param nbSimulation nombre de simulations à effectuer
73       * @param width        largeur de la grille à afficher
74       * @param height       hauteur de la grille à afficher
75       * @param nbVampires   nombe de vampires
76       * @param nbHumans     nombe de humains
77       * @param displayer    affichage de la simulation
78       */
79      Simulator(unsigned nbSimulation, unsigned width, unsigned height, unsigned nbVampires,
80                unsigned nbHumans, Displayer& displayer);
81
82      /**
83       * Destructeur par défaut
84       */
85      ~Simulator() = default;
86
87      /**
88       * @return le nombre de simulations à effectuer
89       */
90      unsigned getNbSimulation() const;
91
92      /**
93       * Lance une session graphique de simulation avec le displayer passé dans le constructeur
94       */
95      void run();
96
97      /**
98       * Simule des parties sur le nombre de simulations choisies et retourne le nombre de sauvetages
99       * sur le nombre de simulations effectuées
100      * @return le nombre de sauvetages sur le nombre de simulations effectuées
101      */
102     double simulate() const;
103
104 private:
105     unsigned nbSimulation, width, height, nbVampires, nbHumans;
106     Field field;
107     Displayer& displayer;
108     bool ended;
109 };
110
111 #endif //BUFFY_SIMULATOR_HPP
112
113 /* ----------------------------------------------------------------------------------------------*/
114
115 /**
116  * Classe contrôlant la simulation du jeu
117  * @author Jonathan Friedli
118  * @author Lazar Pavicevic
119  */
120
121 #include "Simulator.hpp"
122
123 using namespace std;
124
125 Simulator::Simulator(unsigned nbSimulation, unsigned width, unsigned height, unsigned
126 nbVampires, unsigned nbHumans, Displayer& displayer) : nbSimulation(nbSimulation),
127                                                        width(width), height(height),
128                                                        nbVampires(nbVampires),
129                                                        nbHumans(nbHumans),
130                                                        field(width, height, nbVampires, nbHumans),
131                                                        displayer(displayer), ended(false) {
132 }
133
134
135
136
137
138
```

```
139  void Simulator::run() {
140     while (!ended) {
141        displayer.displayGrid(field.begin(), field.end());
142        displayer.displayPrompt(field.getTurn());
143        ended = displayer.getInput(field, *this);
144     }
145  }
146
147  double Simulator::simulate() const {
148     int nbSaves = 0;
149     for (unsigned i = 0; i < nbSimulation; ++i) {
150        Field simulationField(width, height, nbVampires, nbHumans);
151        while (simulationField.hasVampires()) {
152           simulationField.nextTurn();
153        }
154        if (simulationField.hasHumans()) {
155           ++nbSaves;
156        }
157     }
158     return nbSaves / (double)nbSimulation;
159  }
160
161  unsigned Simulator::getNbSimulation() const {
162     return nbSimulation;
163  }
164
165  /* ------------------------------------------------------------------------------------------------*/
166
167  #ifndef BUFFY_FIELD_HPP
168  #define BUFFY_FIELD_HPP
169
170  #include <list>
171  #include <limits>
172  #include "../humanoids/Humanoid.hpp"
173  #include "../humanoids/Vampire.hpp"
174
175  /**
176   * Classe représentant le terrain de jeu contenant une liste d'humanoïdes.
177   * @author Jonathan Friedli
178   * @author Lazar Pavicevic
179   */
180  class Field {
181  public:
182     /**
183      * Constructeur de base
184      * @param width       valeur maximale pour la position x des humanoïdes
185      * @param height      valeur maximale pour la position y des humanoïdes
186      * @param nbVampires  nombre de vampires
187      * @param nbHumans    nombe de humains
188      * @throws runtime_error si les valeurs inserees sont nulles
189      */
190     Field(unsigned width, unsigned height, unsigned nbVampires, unsigned nbHumans);
191
192     /**
193      * Destructeur s'assurant de supprimer les humanoïdes de la liste
194      */
195     ~Field();
196
197     /**
198      * Désactive le constructeur de copie
199      * @param other field à copier
200      */
201     Field(const Field& other) = delete;
202
203     /**
204      * Désactive l'opérateur d'affectation
205      * @param other field à affecter
206      */
207     Field &operator=(const Field& other) = delete;
```

```
208
209     /**
210      * Joue un tour et effectue l'affectation des actions, leurs exécutions
211      * et la suppression des humanoïdes
212      * @return le nombre de tours effectués
213      */
214     int nextTurn();
215
216     /**
217      * @return la valeur maximale pour la position x des humanoïdes
218      */
219     unsigned getWidth() const;
220
221     /**
222      * @return la valeur maximale pour la position y des humanoïdes
223      */
224     unsigned getHeight() const;
225
226     /**
227      * @return le nombre de tours effectués
228      */
229     int getTurn() const;
230
231     /**
232      * @return true s'il y a encore des humains sur le terrain
233      */
234     bool hasHumans() const;
235
236     /**
237      * @return true s'il y a encore des vampires sur le terrain
238      */
239     bool hasVampires() const;
240
241     /**
242      * @return un itérateur sur le premier humanoïde de la liste
243      */
244     std::list<Humanoid*>::const_iterator begin() const;
245
246     /**
247      * @return un itérateur sur le dernier humanoïde de la liste
248      */
249     std::list<Humanoid*>::const_iterator end() const;
250
251     /**
252      * Retourne l'humanoïde avec l'identifiant voulu le plus proche de la position donnée
253      * @tparam T   Type de l'humanoïde recherché
254      * @param from Position de départ
255      * @return l'humanoïde le plus proche de la position donnée
256      */
257     template<typename T>
258     T* getNearestHumanoid(Position& from) const;
259
260     /**
261      * Réduit le nombre des humanoïdes spécifiés
262      * @param target l'humanoïde dont il faut diminuer la population
263      */
264     void decreasePopulation(Humanoid* target);
265
266     /**
267      * Ajoute un vampire à la liste
268      * @param vampire le vampire à ajouter
269      */
270     void addVampire(Vampire* vampire);
271
272 private:
273     unsigned width, height, nbVampires, nbHumans;
274     int turn;
275     std::list<Humanoid*> humanoids;
276 };
```

```

#include "Field_Impl.hpp"

#endif //BUFFY_FIELD_HPP

/* ------------------------------------------------------------------------------------------------*/

/**
 * Fichier contenant les implémentations génériques de la méthode de Field.
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
template<typename T>
T* Field::getNearestHumanoid(Position& from) const {
   double shortestEuclideanDistance = std::numeric_limits<double>::max();
   T* nearestHumanoid = nullptr;
   T* temp;
   for (auto humanoid: humanoids) {
      if ((temp = dynamic_cast<T*>(humanoid)) != nullptr) {
         double euclideanDistance = Position::getEuclideanDistance(from,
                                                                   humanoid->getPosition());
         if (euclideanDistance < shortestEuclideanDistance) {
            shortestEuclideanDistance = euclideanDistance;
            nearestHumanoid = temp;
         }
      }
   }
   return nearestHumanoid;
}

/* ------------------------------------------------------------------------------------------------*/

/**
 * Classe représentant le terrain de jeu contenant une liste d'humanoïdes.
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include <stdexcept>

#include "Field.hpp"
#include "../humanoids/Human.hpp"
#include "../humanoids/Buffy.hpp"
#include "utils/Random.hpp"

using namespace std;

Field::Field(unsigned width, unsigned height, unsigned nbVampires, unsigned nbHumans)
   : width(width), height(height), nbVampires(nbVampires), nbHumans(nbHumans), turn(0) {
   if (width == 0 || height == 0 || nbHumans == 0 || nbVampires == 0) {
      throw runtime_error("Erreur: Les valeurs inserees ne peuvent pas etre nulles");
   }

   for (unsigned i = 0; i < nbHumans; i++) {
      humanoids.emplace_back(new Human((int)Random::random(width), (int)Random::random(height)));
   }

   for (unsigned i = 0; i < nbVampires; i++) {
      humanoids.emplace_back(new Vampire((int)Random::random(width), (int)Random::random(height)));
   }

   humanoids.emplace_back(new Buffy((int)Random::random(width), (int)Random::random(height)));
}
```

```cpp
Field::~Field() {
   for (auto& humanoid: humanoids) {
      delete humanoid;
   }
}

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

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

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

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

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

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

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

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

void Field::decreasePopulation(Humanoid* target) {
   if (dynamic_cast<Human*>(target) != nullptr) {
      nbHumans--;
   } else if (dynamic_cast<Vampire*>(target) != nullptr) {
      nbVampires--;
   }
}

void Field::addVampire(Vampire* vampire) {
   humanoids.emplace_back(vampire);
   nbVampires++;
}

/* ------------------------------------------------------------------------------------------*/
```

```cpp
#ifndef BUFFY_DISPLAYER_HPP
#define BUFFY_DISPLAYER_HPP

#include <list>
#include "../humanoids/Humanoid.hpp"

class Simulator;

/**
 * Classe abstraite gérant l'affichage d'une simulation
 *
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Displayer {
public:
    /**
     * Constructeur de base
     * @param width  largeur de la grille
     * @param height hauteur de la grille
     */
    Displayer(unsigned width, unsigned height);

    /**
     * Destructeur virtuel par défaut
     */
    virtual ~Displayer() = default;

    /**
     * Affiche la grille de la simulation
     * @param begin Itérateur sur le premier élément de la liste des humanoïdes
     * @param end   Itérateur sur le dernier élément de la liste des humanoïdes
     */
    virtual void displayGrid(std::list<Humanoid*>::const_iterator begin,
                             std::list<Humanoid*>::const_iterator end) = 0;

    /**
     * Affiche le prompt
     * @param turn Nombre de tours effectués
     */
    virtual void displayPrompt(int turn) = 0;

    /**
     * Récupère de l'input de l'utilisateur
     * @param field     Field contenant les humanoïdes
     * @param simulator Simulateur actuel
     * @return true si l'utilisateur souhaite quitter la simulation
     */
    virtual bool getInput(Field& field, Simulator& simulator) = 0;

protected:
    /**
     * @return la largeur de la grille
     */
    unsigned getWidth() const;

    /**
     * @return la hauteur de la grille
     */
    unsigned getHeight() const;

private:
    const unsigned width, height;
};

#endif //BUFFY_DISPLAYER_HPP

/* ---------------------------------------------------------------------------------------------*/
```

```cpp
/**
 * Classe abstraite gérant l'affichage d'une simulation
 *
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include "Displayer.hpp"

using namespace std;

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

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

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

/* ------------------------------------------------------------------------------------------*/

#ifndef BUFFY_CONSOLEDISPLAYER_HPP
#define BUFFY_CONSOLEDISPLAYER_HPP

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

/**
 * Classe gérant l'affichage d'une simulation dans une console
 *
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class ConsoleDisplayer : public Displayer {
public:
   /**
    * Constructeur de base
    * @param width  largeur de la grille
    * @param height hauteur de la grille
    */
   ConsoleDisplayer(unsigned width, unsigned height);

   void displayGrid(std::list<Humanoid*>::const_iterator begin,
                    std::list<Humanoid*>::const_iterator end) override;

   void displayPrompt(int turn) override;

   bool getInput(Field& f, Simulator& s) override;

private:
   /**
    * Met à jour la grille en itérant sur la liste des humanoïdes
    * @param begin Itérateur sur le premier élément de la liste des humanoïdes
    * @param end   Itérateur sur le dernier élément de la liste des humanoïdes
    */
   void updateGrid(std::list<Humanoid*>::const_iterator begin,
                   std::list<Humanoid*>::const_iterator end);

   std::vector<std::vector<char>> grid;
};

#endif //BUFFY_CONSOLEDISPLAYER_HPP
```

```cpp
/**
 * Classe gérant l'affichage d'une simulation dans une console
 *
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include <iostream>
#include "../simulation/Field.hpp"
#include "../simulation/Simulator.hpp"
#include "ConsoleDisplayer.hpp"

using namespace std;

ConsoleDisplayer::ConsoleDisplayer(unsigned width, unsigned height)
   : Displayer(width, height), grid(height, vector<char>(width, ' ')) {
}

void ConsoleDisplayer::displayGrid(list<Humanoid*>::const_iterator begin,
                                   list<Humanoid*>::const_iterator end) {
   updateGrid(begin, end);
   cout << "+" << string(getWidth(), '-') << "+" << endl;
   for (auto& row: grid) {
      cout << "|";
      for (auto& humanoid: row) {
         cout << humanoid;
      }
      cout << "|" << endl;
   }
   cout << "+" << string(getWidth(), '-') << "+" << endl;
}

void ConsoleDisplayer::displayPrompt(int turn) {
   cout << "[" << turn << "] q)uit s)tatistics n)ext:";
}

bool ConsoleDisplayer::getInput(Field& f, Simulator& s) {
   string input;
   getline(cin, input);
   if (input == "q") {
      return true;
   } else if (input == "s") {
      cout << "Simulating..." << endl;
      double saves = s.simulate();
      cout << "For " << s.getNbSimulation() << " simulations : succes rate of " << saves * 100
           << "%" << endl;
   } else if (input == "n" || input.empty()) {
      f.nextTurn();
   } else {
      cout << "Invalid input" << endl;
   }
   return false;
}

void ConsoleDisplayer::updateGrid(list<Humanoid*>::const_iterator begin,
                                  list<Humanoid*>::const_iterator end) {
   grid.assign(getHeight(), vector<char>(getWidth(), ' '));
   for (auto iter = begin; iter != end; ++iter) {
      grid.at((size_t)(*iter)->getPosition().getY())
         .at((size_t)(*iter)->getPosition().getX()) = (*iter)->getSymbol();
   }
}

/* ----------------------------------------------------------------------------------------*/
```

```hpp
#ifndef BUFFY_HUMANOID_HPP
#define BUFFY_HUMANOID_HPP

#include "../actions/Action.hpp"
#include "../simulation/utils/Position.hpp"

/**
 * Classe abstraite représentant un humanoïde
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Humanoid {
public:

    /**
     * Constructeur de base
     * @param x Position sur l'axe des x
     * @param y Position sur l'axe des y
     */
    Humanoid(int x, int y);

    /**
     * Destructeur virtual s'assurant de supprimer l'action
     */
    virtual ~Humanoid();

    /**
     * Désactive le constructeur de copie
     * @param other humanoïde à copier
     */
    Humanoid(const Humanoid& other) = delete;

    /**
     * Désactive l'opérateur d'affectation
     * @param other humanoïde à affecter
     */
    Humanoid& operator=(const Humanoid& other) = delete;

    /**
     * Retourne la position de l'humanoïde
     * @return une référence sur la position
     */
    Position& getPosition();

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

    /**
     * Définit l'état de l'huanoïde
     * @param isAlive nouvel état de l'humanoïde
     */
    void setAlive(bool isAlive);

    /**
     * Exécute l'action de l'humanoïde
     * @param field Field sur lequel l'action est effectuée
     */
    void executeAction(Field& field);

    /**
     * Définit l'action de l'humanoïde en fonction du contenu du field
     * @param field Field sur lequel l'action est définie
     */
    virtual void setAction(const Field& field);
```

```
691      /**
692       * @return un symbole représentant l'humanoïde
693       */
694      virtual char getSymbol() const = 0;
695
696   protected:
697      static const int RANGE = 1;
698
699      /**
700       * Setter protected pour définir l'attribut newAction
701       * @param newAction nouvelle newAction
702       */
703      void setAction(Action* newAction);
704
705   private:
706      Action* action;
707      bool alive;
708      Position position;
709   };
710
711   #endif //BUFFY_HUMANOID_HPP
712
713   /* ----------------------------------------------------------------------------------------------------*/
714
715   /**
716    * Classe abstraite représentant un humanoïde.
717    * @author Jonathan Friedli
718    * @author Lazar Pavicevic
719    */
720
721   #include "Humanoid.hpp"
722
723   Humanoid::Humanoid(int x, int y) : action(nullptr), alive(true), position(x, y) {
724   }
725
726   Humanoid::~Humanoid() {
727      delete action;
728   }
729
730   Position& Humanoid::getPosition() {
731      return position;
732   }
733
734   bool Humanoid::isAlive() const {
735      return alive;
736   }
737
738   void Humanoid::setAlive(bool isAlive) {
739      this->alive = isAlive;
740   }
741
742   void Humanoid::executeAction(Field& field) {
743      if (action != nullptr) {
744         action->execute(field);
745      }
746   }
747
748   void Humanoid::setAction(const Field& field) {
749      if (action != nullptr) {
750         delete action;
751         action = nullptr;
752      }
753   }
754
755   void Humanoid::setAction(Action* newAction) {
756      this->action = newAction;
757   }
758
759   /* ----------------------------------------------------------------------------------------------------*/
```

```cpp
#ifndef BUFFY_BUFFY_HPP
#define BUFFY_BUFFY_HPP

#include "Humanoid.hpp"

/**
 * Classe représentant Buffy, la chasseuse de vampire !
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Buffy : public Humanoid {
public:
   /**
    * Constructeur de base
    * @param x Position sur l'axe des x
    * @param y Position sur l'axe des y
    */
   Buffy(int x, int y);

   void setAction(const Field& field) override;

   char getSymbol() const override;
};

#endif //BUFFY_BUFFY_HPP

/* ------------------------------------------------------------------------------------------------*/

/**
 * Classe représentant Buffy, la chasseuse de vampire !
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include "Buffy.hpp"
#include "../simulation/Field.hpp"
#include "../actions/MoveAction.hpp"
#include "../actions/KillAction.hpp"

Buffy::Buffy(int x, int y) : Humanoid(x, y) {
}

void Buffy::setAction(const Field& field) {
   Humanoid::setAction(field);
   if (field.hasVampires()) {
      auto target = field.getNearestHumanoid<Vampire>(getPosition());
      double distance = Position::getEuclideanDistance(getPosition(), target->getPosition());
      if (distance > RANGE) {
         Humanoid::setAction(new MoveAction(*this, target, 2));
      } else {
         Humanoid::setAction(new KillAction(target));
      }
   } else {
      Humanoid::setAction(new MoveAction(*this, nullptr, RANGE));
   }
}

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

/* ------------------------------------------------------------------------------------------------*/
```

```hpp
#ifndef BUFFY_HUMAN_HPP
#define BUFFY_HUMAN_HPP

#include "Humanoid.hpp"

/**
 * Classe représentant un humain
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Human : public Humanoid {
public:
   /**
    * Constructeur de base
    * @param x Position sur l'axe des x
    * @param y Position sur l'axe des y
    */
   Human(int x, int y);

   void setAction(const Field& field) override;

   char getSymbol() const override;
};

#endif //BUFFY_HUMAN_HPP
```

/* ---------- */

```cpp
/**
 * Classe représentant un humain
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include "Human.hpp"
#include "../actions/MoveAction.hpp"

Human::Human(int x, int y) : Humanoid(x, y) {
}

void Human::setAction(const Field& field) {
   Humanoid::setAction(field);
   Humanoid::setAction(new MoveAction(*this, nullptr, RANGE));
}

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

/* ---------- */

```hpp
#ifndef BUFFY_VAMPIRE_HPP
#define BUFFY_VAMPIRE_HPP

#include "Humanoid.hpp"

/**
 * Classe représentant un vampire
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Vampire : public Humanoid {
public:
   /**
    * Constructeur de base
    * @param x Position sur l'axe des x
    * @param y Position sur l'axe des y
    */
   Vampire(int x, int y);
```

```
898
899     void setAction(const Field& field) override;
900
901     char getSymbol() const override;
902 };
903
904 #endif //BUFFY_VAMPIRE_HPP
905
906 /* ---------------------------------------------------------------------------------------------*/
907
908 /**
909  * Classe représentant un vampire
910  * @author Jonathan Friedli
911  * @author Lazar Pavicevic
912  */
913
914 #include "Vampire.hpp"
915 #include "../simulation/Field.hpp"
916 #include "../actions/MoveAction.hpp"
917 #include "../actions/KillAction.hpp"
918 #include "../actions/TransformAction.hpp"
919 #include "../simulation/utils/Random.hpp"
920 #include "Human.hpp"
921
922 Vampire::Vampire(int x, int y) : Humanoid(x, y) {
923 }
924
925 void Vampire::setAction(const Field& field) {
926     Humanoid::setAction(field);
927     if (field.hasHumans()) {
928         auto target = field.getNearestHumanoid<Human>(getPosition());
929         double distance = Position::getEuclideanDistance(getPosition(), target->getPosition());
930         if (distance > RANGE) {
931             Humanoid::setAction(new MoveAction(*this, target, RANGE));
932         } else {
933             if (Random::random(2)) {
934                 Humanoid::setAction(new KillAction(target));
935             } else {
936                 Humanoid::setAction(new TransformAction(target));
937             }
938         }
939     }
940 }
941
942 char Vampire::getSymbol() const {
943     return 'V';
944 }
945
946 /* ---------------------------------------------------------------------------------------------*/
947
948 #ifndef BUFFY_ACTION_HPP
949 #define BUFFY_ACTION_HPP
950
951 class Field;
952
953 class Humanoid;
954
955 /**
956  * Classe représentant une action effectuée par un humanoïde
957  * @author Jonathan Friedli
958  * @author Lazar Pavicevic
959  */
960 class Action {
961 public:
962     /**
963      * Constructeur de base
964      * @param target      la cible de l'action
965      */
966     explicit Action(Humanoid* target);
```

```cpp

    /**
     * Destructeur virtuel par défaut
     */
    virtual ~Action() = default;

    /**
     * Exécute l'action sur le field correspondant
     * @param field Field actuel de la simulation
     */
    virtual void execute(Field& field) = 0;

protected:
    /**
     * @return un pointeur sur la cible de l'action
     */
    Humanoid* getTarget() const;

private:
    Humanoid* target;
};

#endif //BUFFY_ACTION_HPP
/* ---------------------------------------------------------------------------------------------*/

/**
 * Classe représentant une action effectuée par un humanoïde
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include "Action.hpp"

Action::Action(Humanoid* target) : target(target) {
}

Humanoid* Action::getTarget() const {
    return target;
}
/* ---------------------------------------------------------------------------------------------*/
#ifndef BUFFY_MOVEACTION_HPP
#define BUFFY_MOVEACTION_HPP

#include "Action.hpp"

/**
 * Classe représentant une action déplaçant un humanoïde aléatoirement ou vers une cible
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class MoveAction : public Action {
public:
    /**
     * Constructeur de base
     * @param actionMaker l'humanoïde qui effectue l'action
     * @param target      la cible de l'action
     * @param step        le nombre de pas possibles
     */
    MoveAction(Humanoid& actionMaker, Humanoid* target, unsigned step);

    void execute(Field& f) override;

private:
    Humanoid& actionMaker;
    unsigned step;
    int currentX, currentY;
};

#endif //BUFFY_MOVEACTION_HPP
```

```

/* ------------------------------------------------------------------------------------------*/

/**
 * Classe représentant une action déplaçant un humanoïde aléatoirement ou vers une cible
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include "MoveAction.hpp"
#include "../simulation/Field.hpp"

MoveAction::MoveAction(Humanoid& actionMaker, Humanoid* target, unsigned step)
   : Action(target), actionMaker(actionMaker), step(step), currentX(0), currentY(0) {
   if (target != nullptr) {
      currentX = target->getPosition().getX();
      currentY = target->getPosition().getY();
   }
}

void MoveAction::execute(Field& f) {
   if (step != 0) {
      if (getTarget() != nullptr) {
         Position targetPosition(currentX, currentY);
         actionMaker.getPosition().setDirectedPosition(targetPosition, (int)step);
      } else {
         actionMaker.getPosition().setRandomPosition((int)f.getWidth(), (int)f.getHeight());
      }
   }
}

/* ------------------------------------------------------------------------------------------*/


#ifndef BUFFY_KILLACTION_HPP
#define BUFFY_KILLACTION_HPP

#include "Action.hpp"

/**
 * Classe représentant une action qui tue un humanoïde
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class KillAction : public Action {
public:
   /**
    * Constructeur de base
    * @param target      la cible de l'action
    */
   explicit KillAction(Humanoid* target);

   void execute(Field& f) override;
};

#endif //BUFFY_KILLACTION_HPP

/* ------------------------------------------------------------------------------------------*/

/**
 * Classe représentant une action qui tue un humanoïde
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include "KillAction.hpp"
#include "../simulation/Field.hpp"


```

```
KillAction::KillAction(Humanoid* target) : Action(target) {
}

void KillAction::execute(Field& f) {
   if (getTarget()->isAlive()) {
      getTarget()->setAlive(false);
      f.decreasePopulation(getTarget());
   }
}

/* ----------------------------------------------------------------------------------------------*/

#ifndef BUFFY_TRANSFORMACTION_HPP
#define BUFFY_TRANSFORMACTION_HPP

#include "Action.hpp"

/**
 * Classe représentant une action qui transforme un humanoïde en vampire
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class TransformAction : public Action {
public:
   /**
    * Constructeur de base
    * @param target      la cible de l'action
    */
   explicit TransformAction(Humanoid* target);

   void execute(Field& f) override;
};

#endif //BUFFY_TRANSFORMACTION_HPP

/* ----------------------------------------------------------------------------------------------*/

/**
 * Classe représentant une action qui transforme un humanoïde en vampire
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

#include "TransformAction.hpp"
#include "../simulation/Field.hpp"

TransformAction::TransformAction(Humanoid* target) : Action(target) {
}

void TransformAction::execute(Field& f) {
   if (getTarget()->isAlive()) {
      getTarget()->setAlive(false);
      f.decreasePopulation(getTarget());
      f.addVampire(new Vampire(getTarget()->getPosition().getX(),
                               getTarget()->getPosition().getY()));
   }
}
/* ----------------------------------------------------------------------------------------------*/

#ifndef BUFFY_POSITION_HPP
#define BUFFY_POSITION_HPP

/**
 * Classe représentant une position dans le field
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Position {
public:
```

```
1174      /**
1175       * Retourne la distance euclidienne entre deux positions
1176       * @param from Position de départ
1177       * @param to   Position de la destination
1178       * @return un double représentant la distance euclidienne entre deux positions
1179       */
1180      static double getEuclideanDistance(Position& from, Position& to);
1181
1182      /**
1183       * Constructeur de base
1184       * @param x Position sur l'axe des x
1185       * @param y Position sur l'axe des y
1186       */
1187      Position(int x, int y);
1188
1189      /**
1190       * @return la position sur l'axe des x
1191       */
1192      int getX() const;
1193
1194      /**
1195       * @return la position sur l'axe des y
1196       */
1197      int getY() const;
1198
1199      /**
1200       * Définit une nouvelle position aléatoire
1201       * @param maxX Valeur maximale sur l'axe des x
1202       * @param maxY Valeur maximale sur l'axe des y
1203       */
1204      void setRandomPosition(int maxX, int maxY);
1205
1206      /**
1207       * Définit une nouvelle position se rapprochant de la position donnée
1208       * @param target Position cible
1209       * @param step   Nombre de pas possibles
1210       */
1211      void setDirectedPosition(Position& target, int step);
1212
1213  private :
1214      int x, y;
1215  };
1216
1217  #endif //BUFFY_POSITION_HPP
1218
1219  /* ----------------------------------------------------------------------------------------------------*/
1220
1221  /**
1222   * Classe représentant une position dans le field
1223   * @author Jonathan Friedli
1224   * @author Lazar Pavicevic
1225   */
1226
1227  #include <cmath>
1228  #include "Direction.hpp"
1229  #include "Random.hpp"
1230
1231  double Position::getEuclideanDistance(Position& from, Position& to) {
1232      double first = abs((from.getX() - to.getX()));
1233      double second = abs((from.getY() - to.getY()));
1234      return round(hypot(first, second));
1235  }
1236
1237  Position::Position(int x, int y) : x(x), y(y) {
1238  }
1239
1240  int Position::getX() const {
1241      return x;
1242  }
```

```cpp

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

void Position::setRandomPosition(int maxX, int maxY) {
   std::vector<const Direction*> possibleDirections;
   bool isRestricted = false;

   if (x == 0 || y == 0 || x == maxX - 1 || y == maxY - 1) {
      isRestricted = true;
      for (const Direction* s = *Direction::values();
           s <= Direction::values()[Direction::size() - 1]; s++) {
         if (x + s->getX() >= 0 && y + s->getY() >= 0 && x + s->getX() < maxX &&
             y + s->getY() < maxY) {
            possibleDirections.emplace_back(s);
         }
      }
   }

   auto& direction = isRestricted
                     ? *possibleDirections[Random::random((unsigned)possibleDirections.size())]
                     : Direction::get(Random::random(Direction::size()));
   x += direction.getX();
   y += direction.getY();
}

void Position::setDirectedPosition(Position& target, int step) {
   int newX = x, newY = y;

   for (int i = 0; i < step; i++) {
      int dirX = target.getX() - newX;
      int dirY = target.getY() - newY;

      dirX = dirX == 0 ? 0 : dirX / (abs(dirX));
      dirY = dirY == 0 ? 0 : dirY / (abs(dirY));

      newX += dirX;
      newY += dirY;
   }
   x = newX;
   y = newY;
}

/* ----------------------------------------------------------------------------------------------------*/

#ifndef BUFFY_DIRECTION_HPP
#define BUFFY_DIRECTION_HPP

/**
 * Enum représentant les directions possibles d'un humanoïde
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Direction {
public:

   /**
    * Retourne la direction correspondant à l'index donné
    * @param index index de la direction
    * @throws std::out_of_range si l'index est trop grand
    * @return la direction correspondant à l'index donné
    */
   static const Direction& get(unsigned index);

   /**
    * @return le nombre de directions possibles
    */
   static unsigned size();
```

```
1312
1313     /**
1314      * @return un tableau de pointeurs sur les directions possibles
1315      */
1316     static const Direction** values();
1317
1318     /**
1319      * @return la direction sur l'axe des x
1320      */
1321     int getX() const;
1322
1323     /**
1324      * @return la direction sur l'axe des y
1325      */
1326     int getY() const;
1327
1328     /**
1329      * Constructeur de copie inaccessible
1330      */
1331     Direction(const Direction& other) = delete;
1332
1333     /**
1334      * Opérateur d'affectation inaccessible
1335      */
1336     Direction& operator=(const Direction& other) = delete;
1337
1338     static const Direction LEFT_UP, UP, RIGHT_UP, LEFT, RIGHT, LEFT_DOWN, DOWN, RIGHT_DOWN;
1339
1340 private:
1341     /**
1342      * Constructeur de base privé
1343      * @param x Direction sur l'axe des x
1344      * @param y Direction sur l'axe des y
1345      */
1346     Direction(int x, int y);
1347
1348     static unsigned COUNT;
1349     static const Direction* DIRECTIONS[];
1350     int x, y;
1351 };
1352
1353 #endif //BUFFY_DIRECTION_HPP
1354
1355 /* ----------------------------------------------------------------------------------------------*/
1356
1357 /**
1358  * Enum représentant les directions possibles d'un humanoïde
1359  * @author Jonathan Friedli
1360  * @author Lazar Pavicevic
1361  */
1362
1363 #include <stdexcept>
1364 #include "Direction.hpp"
1365
1366 const Direction Direction::LEFT_UP(-1, -1);
1367 const Direction Direction::UP(0, -1);
1368 const Direction Direction::RIGHT_UP(1, -1);
1369 const Direction Direction::LEFT(-1, 0);
1370 const Direction Direction::RIGHT(1, 0);
1371 const Direction Direction::LEFT_DOWN(-1, 1);
1372 const Direction Direction::DOWN(0, 1);
1373 const Direction Direction::RIGHT_DOWN(1, 1);
1374
1375 unsigned Direction::COUNT = 0;
1376
1377 const Direction* Direction::DIRECTIONS[]{
1378     &LEFT_UP, &UP, &RIGHT_UP, &LEFT, &RIGHT, &LEFT_DOWN, &DOWN, &RIGHT_DOWN
1379 };
1380
```

```
const Direction& Direction::get(unsigned index) {
   if (index >= COUNT) {
      throw std::out_of_range("Erreur: L'index est trop grand");
   }
   return *DIRECTIONS[index];
}

unsigned Direction::size() {
   return COUNT;
}

const Direction** Direction::values() {
   return DIRECTIONS;
}

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

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

Direction::Direction(int x, int y) {
   this->x = x;
   this->y = y;
   COUNT++;
}
/* ----------------------------------------------------------------------------------------------*/

#ifndef BUFFY_RANDOM_HPP
#define BUFFY_RANDOM_HPP

#include <string>
#include <random>
#include "../../humanoids/Humanoid.hpp"

/**
 * Classe helper générant des nombres aléatoires
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */
class Random {
public:
   /**
    * Retourne un entier aléatoire entre 0 et la valeur maximale non-comprise
    * @param max Valeur maximale
    * @return un entier aléatoire entre 0 et la valeur maximale non-comprise
    */
   static unsigned random(unsigned max);

   /**
    * Constructeur de copie inaccessible
    */
   Random(Random& other) = delete;

   /**
    * Opérateur d'affectation inaccessible
    */
   void operator=(Random& other) = delete;

private:
   /**
    * Constructeur de base privé
    */
   Random() = default;

   static std::mt19937 engine;
};
```

```
#endif //BUFFY_RANDOM_HPP

/* ----------------------------------------------------------------------------------------------------*/


/**
 * Classe helper générant des nombres aléatoires
 * @author Jonathan Friedli
 * @author Lazar Pavicevic
 */

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

using namespace std;

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

unsigned Random::random(unsigned max) {
   uniform_int_distribution<unsigned> distribution(0, max - 1);
   return distribution(engine);
}
```