

```

1  /**
2   * Classe représentant le point d'entrée du programme
3   * @author Jonathan Friedli
4   * @author Lazar Pavicevic
5   */
6
7  #include <iostream>
8  #include "simulation/Simulator.hpp"
9  #include "displayers/ConsoleDisplayer.hpp"
10
11  #define NB_ARGS 4
12  #define NB_SIMULATION 10000
13
14  int main(int argc, char* argv[]) {
15      unsigned values[NB_ARGS];
16
17      if (argc != NB_ARGS + 1) {
18          throw std::invalid_argument("Usage : buffy <largeur grille> <hauteur grille> <nb vampires>"
19                                     " <nb humains>");
20      }
21      try{
22          for (unsigned i = 1; i < NB_ARGS + 1; ++i) {
23              if (std::stoi(argv[i]) <= 0) {
24                  throw std::exception();
25              }
26              values[i - 1] = (unsigned)std::stoi(argv[i]);
27          }
28      }catch (const std::exception&){ // Permet de catch les exceptions de std::stoi
29          // et si la valeur est < 1
30          throw std::invalid_argument("Les arguments doivent etre des nombres entiers positifs !");
31      }
32
33
34      unsigned width = values[0], height = values[1], nbVampires = values[2], nbHumans = values[3];
35      ConsoleDisplayer displayer(width, height);
36
37      Simulator simulator(NB_SIMULATION, width, height, nbVampires, nbHumans, displayer);
38      simulator.run();
39
40      return EXIT_SUCCESS;
41  }
42
43  /* -----*/
44
45  #ifndef BUFFY_SIMULATOR_HPP
46  #define BUFFY_SIMULATOR_HPP
47
48  #include <string>
49  #include "../displayers/Displayer.hpp"
50  #include "Field.hpp"
51  #include "utils/Random.hpp"
52
53  /**
54   * Classe contrôlant la simulation du jeu
55   * @author Jonathan Friedli
56   * @author Lazar Pavicevic
57   */
58  class Simulator {
59  public:
60
61
62
63
64
65
66
67
68
69

```

```

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    nombre de vampires
76  * @param nbHumans      nombre 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   nombre de humains
188      * @throws runtime_error si les valeurs insérées 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 };

```

```

277
278 #include "Field_Impl.hpp"
279
280 #endif //BUFFY_FIELD_HPP
281
282 /* ----- */
283
284 /**
285  * Fichier contenant les implémentations génériques de la méthode de Field.
286  * @author Jonathan Friedli
287  * @author Lazar Pavicevic
288  */
289 template<typename T>
290 T* Field::getNearestHumanoid(Position& from) const {
291     double shortestEuclideanDistance = std::numeric_limits<double>::max();
292     T* nearestHumanoid = nullptr;
293     T* temp;
294     for (auto humanoid: humanoids) {
295         if ((temp = dynamic_cast<T*>(humanoid)) != nullptr) {
296             double euclideanDistance = Position::getEuclideanDistance(from,
297                                                                     humanoid->getPosition());
298             if (euclideanDistance < shortestEuclideanDistance) {
299                 shortestEuclideanDistance = euclideanDistance;
300                 nearestHumanoid = temp;
301             }
302         }
303     }
304     return nearestHumanoid;
305 }
306
307 /* ----- */
308
309 /**
310  * Classe représentant le terrain de jeu contenant une liste d'humanoïdes.
311  * @author Jonathan Friedli
312  * @author Lazar Pavicevic
313  */
314
315 #include <stdexcept>
316
317 #include "Field.hpp"
318 #include "../humanoids/Human.hpp"
319 #include "../humanoids/Buffy.hpp"
320 #include "utils/Random.hpp"
321
322 using namespace std;
323
324 Field::Field(unsigned width, unsigned height, unsigned nbVampires, unsigned nbHumans)
325 : width(width), height(height), nbVampires(nbVampires), nbHumans(nbHumans), turn(0) {
326     if (width == 0 || height == 0 || nbHumans == 0 || nbVampires == 0) {
327         throw runtime_error("Erreur: Les valeurs inserees ne peuvent pas etre nulles");
328     }
329
330     for (unsigned i = 0; i < nbHumans; i++) {
331         humanoids.emplace_back(new Human((int)Random::random(width), (int)Random::random(height)));
332     }
333
334     for (unsigned i = 0; i < nbVampires; i++) {
335         humanoids.emplace_back(new Vampire((int)Random::random(width), (int)Random::random(height)));
336     }
337
338     humanoids.emplace_back(new Buffy((int)Random::random(width), (int)Random::random(height)));
339 }
340
341
342
343
344
345

```

```

346 Field::~~Field() {
347     for (auto& humanoid: humanoids) {
348         delete humanoid;
349     }
350 }
351
352 int Field::nextTurn() {
353     // Déterminer les prochaines actions
354     for (auto& humanoid: humanoids)
355         humanoid->setAction(*this);
356     // Executer les actions
357     for (auto& humanoid: humanoids)
358         humanoid->executeAction(*this);
359     // Enlever les humanoïdes tués
360     for (auto it = humanoids.begin(); it != humanoids.end(); )
361         if (!(*it)->isAlive()) {
362             delete *it; // destruction de l'humanoïde référencé
363             it = humanoids.erase(it); // suppression de l'élément dans la liste
364         } else
365             ++it;
366     return turn++;
367 }
368
369 unsigned Field::getWidth() const {
370     return width;
371 }
372
373 unsigned Field::getHeight() const {
374     return height;
375 }
376
377 int Field::getTurn() const {
378     return turn;
379 }
380
381 bool Field::hasHumans() const {
382     return nbHumans > 0;
383 }
384
385 bool Field::hasVampires() const {
386     return nbVampires > 0;
387 }
388
389 std::list<Humanoid*>::const_iterator Field::begin() const {
390     return humanoids.begin();
391 }
392
393 std::list<Humanoid*>::const_iterator Field::end() const {
394     return humanoids.end();
395 }
396
397 void Field::decreasePopulation(Humanoid* target) {
398     if (dynamic_cast<Human*>(target) != nullptr) {
399         nbHumans--;
400     } else if (dynamic_cast<Vampire*>(target) != nullptr) {
401         nbVampires--;
402     }
403 }
404
405 void Field::addVampire(Vampire* vampire) {
406     humanoids.emplace_back(vampire);
407     nbVampires++;
408 }
409
410 /* -----*/
411
412
413
414

```

```

415 #ifndef BUFFY_DISPLAYER_HPP
416 #define BUFFY_DISPLAYER_HPP
417
418 #include <list>
419 #include "../humanoids/Humanoid.hpp"
420
421 class Simulator;
422
423 /**
424  * Classe abstraite gérant l'affichage d'une simulation
425  *
426  * @author Jonathan Friedli
427  * @author Lazar Pavicevic
428  */
429 class Displayer {
430 public:
431     /**
432      * Constructeur de base
433      * @param width largeur de la grille
434      * @param height hauteur de la grille
435      */
436     Displayer(unsigned width, unsigned height);
437
438     /**
439      * Destructeur virtuel par défaut
440      */
441     virtual ~Displayer() = default;
442
443     /**
444      * Affiche la grille de la simulation
445      * @param begin Itérateur sur le premier élément de la liste des humanoïdes
446      * @param end Itérateur sur le dernier élément de la liste des humanoïdes
447      */
448     virtual void displayGrid(std::list<Humanoid*>::const_iterator begin,
449                             std::list<Humanoid*>::const_iterator end) = 0;
450
451     /**
452      * Affiche le prompt
453      * @param turn Nombre de tours effectués
454      */
455     virtual void displayPrompt(int turn) = 0;
456
457     /**
458      * Récupère de l'input de l'utilisateur
459      * @param field Field contenant les humanoïdes
460      * @param simulator Simulateur actuel
461      * @return true si l'utilisateur souhaite quitter la simulation
462      */
463     virtual bool getInput(Field& field, Simulator& simulator) = 0;
464
465 protected:
466     /**
467      * @return la largeur de la grille
468      */
469     unsigned getWidth() const;
470
471     /**
472      * @return la hauteur de la grille
473      */
474     unsigned getHeight() const;
475
476 private:
477     const unsigned width, height;
478 };
479
480 #endif //BUFFY_DISPLAYER_HPP
481
482 /* ----- */
483

```

```

484  /**
485   * Classe abstraite gérant l'affichage d'une simulation
486   *
487   * @author Jonathan Friedli
488   * @author Lazar Pavicevic
489   */
490
491  #include "Displayer.hpp"
492
493  using namespace std;
494
495  Displayer::Displayer(unsigned width, unsigned height) : width(width), height(height) {
496  }
497
498  unsigned Displayer::getWidth() const {
499      return width;
500  }
501
502  unsigned Displayer::getHeight() const {
503      return height;
504  }
505
506  /* -----*/
507
508  #ifndef BUFFY_CONSOLEDISPLAYER_HPP
509  #define BUFFY_CONSOLEDISPLAYER_HPP
510
511  #include <vector>
512  #include "Displayer.hpp"
513
514  /**
515   * Classe gérant l'affichage d'une simulation dans une console
516   *
517   * @author Jonathan Friedli
518   * @author Lazar Pavicevic
519   */
520  class ConsoleDisplayer : public Displayer {
521  public:
522      /**
523       * Constructeur de base
524       * @param width largeur de la grille
525       * @param height hauteur de la grille
526       */
527      ConsoleDisplayer(unsigned width, unsigned height);
528
529      void displayGrid(std::list<Humanoid*>::const_iterator begin,
530                      std::list<Humanoid*>::const_iterator end) override;
531
532      void displayPrompt(int turn) override;
533
534      bool getInput(Field& f, Simulator& s) override;
535
536  private:
537      /**
538       * Met à jour la grille en itérant sur la liste des humanoïdes
539       * @param begin Itérateur sur le premier élément de la liste des humanoïdes
540       * @param end Itérateur sur le dernier élément de la liste des humanoïdes
541       */
542      void updateGrid(std::list<Humanoid*>::const_iterator begin,
543                      std::list<Humanoid*>::const_iterator end);
544
545      std::vector<std::vector<char>>> grid;
546  };
547
548  #endif //BUFFY_CONSOLEDISPLAYER_HPP
549
550
551
552

```



```

553  /**
554   * Classe gérant l'affichage d'une simulation dans une console
555   *
556   * @author Jonathan Friedli
557   * @author Lazar Pavicevic
558   */
559
560  #include <iostream>
561  #include "../simulation/Field.hpp"
562  #include "../simulation/Simulator.hpp"
563  #include "ConsoleDisplay.hpp"
564
565  using namespace std;
566
567  ConsoleDisplay::ConsoleDisplay(unsigned width, unsigned height)
568      : Display(width, height), grid(height, vector<char>(width, ' ')) {
569  }
570
571  void ConsoleDisplay::displayGrid(list<Humanoid*>::const_iterator begin,
572                                  list<Humanoid*>::const_iterator end) {
573      updateGrid(begin, end);
574      cout << "+" << string(getWidth(), '-') << "+" << endl;
575      for (auto& row: grid) {
576          cout << "|";
577          for (auto& humanoid: row) {
578              cout << humanoid;
579          }
580          cout << "|" << endl;
581      }
582      cout << "+" << string(getWidth(), '-') << "+" << endl;
583  }
584
585  void ConsoleDisplay::displayPrompt(int turn) {
586      cout << "[" << turn << "] q)uit s)tatistics n)ext:";
587  }
588
589  bool ConsoleDisplay::getInput(Field& f, Simulator& s) {
590      string input;
591      getline(cin, input);
592      if (input == "q") {
593          return true;
594      } else if (input == "s") {
595          cout << "Simulating..." << endl;
596          double saves = s.simulate();
597          cout << "For " << s.getNbSimulation() << " simulations : succes rate of " << saves * 100
598              << "%" << endl;
599      } else if (input == "n" || input.empty()) {
600          f.nextTurn();
601      } else {
602          cout << "Invalid input" << endl;
603      }
604      return false;
605  }
606
607  void ConsoleDisplay::updateGrid(list<Humanoid*>::const_iterator begin,
608                                  list<Humanoid*>::const_iterator end) {
609      grid.assign(getHeight(), vector<char>(getWidth(), ' '));
610      for (auto iter = begin; iter != end; ++iter) {
611          grid.at((size_t)(*iter)->getPosition().getY())
612              .at((size_t)(*iter)->getPosition().getX()) = (*iter)->getSymbol();
613      }
614  }
615
616  /* -----*/
617
618
619
620
621

```

```

622 #ifndef BUFFY_HUMANOID_HPP
623 #define BUFFY_HUMANOID_HPP
624
625 #include "../actions/Action.hpp"
626 #include "../simulation/utils/Position.hpp"
627
628 /**
629  * Classe abstraite représentant un humanoïde
630  * @author Jonathan Friedli
631  * @author Lazar Pavicevic
632  */
633 class Humanoid {
634 public:
635
636     /**
637      * Constructeur de base
638      * @param x Position sur l'axe des x
639      * @param y Position sur l'axe des y
640      */
641     Humanoid(int x, int y);
642
643     /**
644      * Destructeur virtual s'assurant de supprimer l'action
645      */
646     virtual ~Humanoid();
647
648     /**
649      * Désactive le constructeur de copie
650      * @param other humanoïde à copier
651      */
652     Humanoid(const Humanoid& other) = delete;
653
654     /**
655      * Désactive l'opérateur d'affectation
656      * @param other humanoïde à affecter
657      */
658     Humanoid& operator=(const Humanoid& other) = delete;
659
660     /**
661      * Retourne la position de l'humanoïde
662      * @return une référence sur la position
663      */
664     Position& getPosition();
665
666     /**
667      * @return true si l'humanoïde est vivant
668      */
669     bool isAlive() const;
670
671     /**
672      * Définit l'état de l'humanoïde
673      * @param isAlive nouvel état de l'humanoïde
674      */
675     void setAlive(bool isAlive);
676
677     /**
678      * Exécute l'action de l'humanoïde
679      * @param field Field sur lequel l'action est effectuée
680      */
681     void executeAction(Field& field);
682
683     /**
684      * Définit l'action de l'humanoïde en fonction du contenu du field
685      * @param field Field sur lequel l'action est définie
686      */
687     virtual void setAction(const Field& field);
688
689
690

```

```

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 /* -----*/

```

```

760 #ifndef BUFFY_BUFFY_HPP
761 #define BUFFY_BUFFY_HPP
762
763 #include "Humanoid.hpp"
764
765 /**
766  * Classe représentant Buffy, la chasseuse de vampire !
767  * @author Jonathan Friedli
768  * @author Lazar Pavicevic
769  */
770 class Buffy : public Humanoid {
771 public:
772     /**
773      * Constructeur de base
774      * @param x Position sur l'axe des x
775      * @param y Position sur l'axe des y
776      */
777     Buffy(int x, int y);
778
779     void setAction(const Field& field) override;
780
781     char getSymbol() const override;
782 };
783
784 #endif //BUFFY_BUFFY_HPP
785
786 /* -----*/
787
788 /**
789  * Classe représentant Buffy, la chasseuse de vampire !
790  * @author Jonathan Friedli
791  * @author Lazar Pavicevic
792  */
793
794 #include "Buffy.hpp"
795 #include "../simulation/Field.hpp"
796 #include "../actions/MoveAction.hpp"
797 #include "../actions/KillAction.hpp"
798
799 Buffy::Buffy(int x, int y) : Humanoid(x, y) {
800 }
801
802 void Buffy::setAction(const Field& field) {
803     Humanoid::setAction(field);
804     if (field.hasVampires()) {
805         auto target = field.getNearestHumanoid<Vampire>(getPosition());
806         double distance = Position::getEuclideanDistance(getPosition(), target->getPosition());
807         if (distance > RANGE) {
808             Humanoid::setAction(new MoveAction(*this, target, 2));
809         } else {
810             Humanoid::setAction(new KillAction(target));
811         }
812     } else {
813         Humanoid::setAction(new MoveAction(*this, nullptr, RANGE));
814     }
815 }
816
817 char Buffy::getSymbol() const {
818     return 'B';
819 }
820
821 /* -----*/
822
823
824
825
826
827
828

```

```

829 #ifndef BUFFY_HUMAN_HPP
830 #define BUFFY_HUMAN_HPP
831
832 #include "Humanoid.hpp"
833
834 /**
835  * Classe représentant un humain
836  * @author Jonathan Friedli
837  * @author Lazar Pavicevic
838  */
839 class Human : public Humanoid {
840 public:
841     /**
842      * Constructeur de base
843      * @param x Position sur l'axe des x
844      * @param y Position sur l'axe des y
845      */
846     Human(int x, int y);
847
848     void setAction(const Field& field) override;
849
850     char getSymbol() const override;
851 };
852
853 #endif //BUFFY_HUMAN_HPP
854
855 /* ----- */
856
857 /**
858  * Classe représentant un humain
859  * @author Jonathan Friedli
860  * @author Lazar Pavicevic
861  */
862
863 #include "Human.hpp"
864 #include "../actions/MoveAction.hpp"
865
866 Human::Human(int x, int y) : Humanoid(x, y) {
867 }
868
869 void Human::setAction(const Field& field) {
870     Humanoid::setAction(field);
871     Humanoid::setAction(new MoveAction(*this, nullptr, RANGE));
872 }
873
874 char Human::getSymbol() const {
875     return 'h';
876 }
877
878 /* ----- */
879
880 #ifndef BUFFY_VAMPIRE_HPP
881 #define BUFFY_VAMPIRE_HPP
882
883 #include "Humanoid.hpp"
884
885 /**
886  * Classe représentant un vampire
887  * @author Jonathan Friedli
888  * @author Lazar Pavicevic
889  */
890 class Vampire : public Humanoid {
891 public:
892     /**
893      * Constructeur de base
894      * @param x Position sur l'axe des x
895      * @param y Position sur l'axe des y
896      */
897     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);

```

```

967
968     /**
969     * Destructeur virtuel par défaut
970     */
971     virtual ~Action() = default;
972
973     /**
974     * Exécute l'action sur le field correspondant
975     * @param field Field actuel de la simulation
976     */
977     virtual void execute(Field& field) = 0;
978
979 protected:
980     /**
981     * @return un pointeur sur la cible de l'action
982     */
983     Humanoid* getTarget() const;
984
985 private:
986     Humanoid* target;
987 };
988
989 #endif //BUFFY_ACTION_HPP
990 /* ----- */
991
992 /**
993 * Classe représentant une action effectuée par un humanoïde
994 * @author Jonathan Friedli
995 * @author Lazar Pavicevic
996 */
997
998 #include "Action.hpp"
999
1000 Action::Action(Humanoid* target) : target(target) {
1001 }
1002
1003 Humanoid* Action::getTarget() const {
1004     return target;
1005 }
1006 /* ----- */
1007 #ifndef BUFFY_MOVEACTION_HPP
1008 #define BUFFY_MOVEACTION_HPP
1009
1010 #include "Action.hpp"
1011
1012 /**
1013 * Classe représentant une action déplaçant un humanoïde aléatoirement ou vers une cible
1014 * @author Jonathan Friedli
1015 * @author Lazar Pavicevic
1016 */
1017 class MoveAction : public Action {
1018 public:
1019     /**
1020     * Constructeur de base
1021     * @param actionMaker l'humanoïde qui effectue l'action
1022     * @param target      la cible de l'action
1023     * @param step        le nombre de pas possibles
1024     */
1025     MoveAction(Humanoid& actionMaker, Humanoid* target, unsigned step);
1026
1027     void execute(Field& f) override;
1028
1029 private:
1030     Humanoid& actionMaker;
1031     unsigned step;
1032     int currentX, currentY;
1033 };
1034
1035 #endif //BUFFY_MOVEACTION_HPP

```

```

1036
1037  /* -----*/
1038
1039  /**
1040   * Classe représentant une action déplaçant un humanoïde aléatoirement ou vers une cible
1041   * @author Jonathan Friedli
1042   * @author Lazar Pavicevic
1043   */
1044
1045  #include "MoveAction.hpp"
1046  #include "../simulation/Field.hpp"
1047
1048  MoveAction::MoveAction(Humanoid& actionMaker, Humanoid* target, unsigned step)
1049      : Action(target), actionMaker(actionMaker), step(step), currentX(0), currentY(0) {
1050      if (target != nullptr) {
1051          currentX = target->getPosition().getX();
1052          currentY = target->getPosition().getY();
1053      }
1054  }
1055
1056  void MoveAction::execute(Field& f) {
1057      if (step != 0) {
1058          if (getTarget() != nullptr) {
1059              Position targetPosition(currentX, currentY);
1060              actionMaker.getPosition().setDirectedPosition(targetPosition, (int)step);
1061          } else {
1062              actionMaker.getPosition().setRandomPosition((int)f.getWidth(), (int)f.getHeight());
1063          }
1064      }
1065  }
1066
1067  /* -----*/
1068
1069  #ifndef BUFFY_KILLACTION_HPP
1070  #define BUFFY_KILLACTION_HPP
1071
1072  #include "Action.hpp"
1073
1074  /**
1075   * Classe représentant une action qui tue un humanoïde
1076   * @author Jonathan Friedli
1077   * @author Lazar Pavicevic
1078   */
1079
1080  class KillAction : public Action {
1081  public:
1082      /**
1083       * Constructeur de base
1084       * @param target la cible de l'action
1085       */
1086      explicit KillAction(Humanoid* target);
1087
1088      void execute(Field& f) override;
1089  };
1090
1091  #endif //BUFFY_KILLACTION_HPP
1092
1093  /* -----*/
1094
1095  /**
1096   * Classe représentant une action qui tue un humanoïde
1097   * @author Jonathan Friedli
1098   * @author Lazar Pavicevic
1099   */
1100
1101  #include "KillAction.hpp"
1102  #include "../simulation/Field.hpp"
1103
1104

```



```

1105 KillAction::KillAction(Humanoid* target) : Action(target) {
1106 }
1107
1108 void KillAction::execute(Field& f) {
1109     if (getTarget()->isAlive()) {
1110         getTarget()->setAlive(false);
1111         f.decreasePopulation(getTarget());
1112     }
1113 }
1114
1115 /* -----*/
1116
1117 #ifndef BUFFY_TRANSFORMACTION_HPP
1118 #define BUFFY_TRANSFORMACTION_HPP
1119
1120 #include "Action.hpp"
1121
1122 /**
1123  * Classe représentant une action qui transforme un humanoïde en vampire
1124  * @author Jonathan Friedli
1125  * @author Lazar Pavicevic
1126  */
1127 class TransformAction : public Action {
1128 public:
1129     /**
1130      * Constructeur de base
1131      * @param target la cible de l'action
1132      */
1133     explicit TransformAction(Humanoid* target);
1134
1135     void execute(Field& f) override;
1136 };
1137
1138 #endif //BUFFY_TRANSFORMACTION_HPP
1139
1140 /* -----*/
1141
1142 /**
1143  * Classe représentant une action qui transforme un humanoïde en vampire
1144  * @author Jonathan Friedli
1145  * @author Lazar Pavicevic
1146  */
1147
1148 #include "TransformAction.hpp"
1149 #include "../simulation/Field.hpp"
1150
1151 TransformAction::TransformAction(Humanoid* target) : Action(target) {
1152 }
1153
1154 void TransformAction::execute(Field& f) {
1155     if (getTarget()->isAlive()) {
1156         getTarget()->setAlive(false);
1157         f.decreasePopulation(getTarget());
1158         f.addVampire(new Vampire(getTarget()->getPosition().getX(),
1159                                 getTarget()->getPosition().getY()));
1160     }
1161 }
1162 /* -----*/
1163
1164 #ifndef BUFFY_POSITION_HPP
1165 #define BUFFY_POSITION_HPP
1166
1167 /**
1168  * Classe représentant une position dans le field
1169  * @author Jonathan Friedli
1170  * @author Lazar Pavicevic
1171  */
1172 class Position {
1173 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 }

```

```

1243
1244 int Position::getY() const {
1245     return y;
1246 }
1247
1248 void Position::setRandomPosition(int maxX, int maxY) {
1249     std::vector<const Direction*> possibleDirections;
1250     bool isRestricted = false;
1251
1252     if (x == 0 || y == 0 || x == maxX - 1 || y == maxY - 1) {
1253         isRestricted = true;
1254         for (const Direction* s = *Direction::values();
1255              s <= Direction::values()[Direction::size() - 1]; s++) {
1256             if (x + s->getX() >= 0 && y + s->getY() >= 0 && x + s->getX() < maxX &&
1257                 y + s->getY() < maxY) {
1258                 possibleDirections.emplace_back(s);
1259             }
1260         }
1261     }
1262
1263     auto& direction = isRestricted
1264         ? *possibleDirections[Random::random((unsigned)possibleDirections.size())]
1265         : Direction::get(Random::random(Direction::size()));
1266     x += direction.getX();
1267     y += direction.getY();
1268 }
1269
1270 void Position::setDirectedPosition(Position& target, int step) {
1271     int newX = x, newY = y;
1272
1273     for (int i = 0; i < step; i++) {
1274         int dirX = target.getX() - newX;
1275         int dirY = target.getY() - newY;
1276
1277         dirX = dirX == 0 ? 0 : dirX / (abs(dirX));
1278         dirY = dirY == 0 ? 0 : dirY / (abs(dirY));
1279
1280         newX += dirX;
1281         newY += dirY;
1282     }
1283     x = newX;
1284     y = newY;
1285 }
1286
1287 /* -----*/
1288
1289 #ifndef BUFFY_DIRECTION_HPP
1290 #define BUFFY_DIRECTION_HPP
1291
1292 /**
1293  * Enum représentant les directions possibles d'un humanoïde
1294  * @author Jonathan Friedli
1295  * @author Lazar Pavicevic
1296  */
1297 class Direction {
1298 public:
1299
1300     /**
1301      * Retourne la direction correspondant à l'index donné
1302      * @param index index de la direction
1303      * @throws std::out_of_range si l'index est trop grand
1304      * @return la direction correspondant à l'index donné
1305      */
1306     static const Direction& get(unsigned index);
1307
1308     /**
1309      * @return le nombre de directions possibles
1310      */
1311     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

```

```

1381     const Direction& Direction::get(unsigned index) {
1382         if (index >= COUNT) {
1383             throw std::out_of_range("Erreur: L'index est trop grand");
1384         }
1385         return *DIRECTIONS[index];
1386     }
1387
1388     unsigned Direction::size() {
1389         return COUNT;
1390     }
1391
1392     const Direction** Direction::values() {
1393         return DIRECTIONS;
1394     }
1395
1396     int Direction::getX() const {
1397         return x;
1398     }
1399
1400     int Direction::getY() const {
1401         return y;
1402     }
1403
1404     Direction::Direction(int x, int y) {
1405         this->x = x;
1406         this->y = y;
1407         COUNT++;
1408     }
1409     /* ----- */
1410
1411     #ifndef BUFFY_RANDOM_HPP
1412     #define BUFFY_RANDOM_HPP
1413
1414     #include <string>
1415     #include <random>
1416     #include "../humanoids/Humanoid.hpp"
1417
1418     /**
1419      * Classe helper générant des nombres aléatoires
1420      * @author Jonathan Friedli
1421      * @author Lazar Pavicevic
1422      */
1423     class Random {
1424     public:
1425         /**
1426          * Retourne un entier aléatoire entre 0 et la valeur maximale non-comprise
1427          * @param max Valeur maximale
1428          * @return un entier aléatoire entre 0 et la valeur maximale non-comprise
1429          */
1430         static unsigned random(unsigned max);
1431
1432         /**
1433          * Constructeur de copie inaccessible
1434          */
1435         Random(Random& other) = delete;
1436
1437         /**
1438          * Opérateur d'affectation inaccessible
1439          */
1440         void operator=(Random& other) = delete;
1441
1442     private:
1443         /**
1444          * Constructeur de base privé
1445          */
1446         Random() = default;
1447
1448         static std::mt19937 engine;
1449     };

```

```
1450
1451 #endif //BUFFY_RANDOM_HPP
1452
1453 /* ----- */
1454
1455 /**
1456  * Classe helper générant des nombres aléatoires
1457  * @author Jonathan Friedli
1458  * @author Lazar Pavicevic
1459  */
1460
1461 #include "Random.hpp"
1462 #include <chrono>
1463
1464 using namespace std;
1465
1466 mt19937 Random::engine((unsigned)chrono::system_clock::now().time_since_epoch().count());
1467
1468 unsigned Random::random(unsigned max) {
1469     uniform_int_distribution<unsigned> distribution(0, max - 1);
1470     return distribution(engine);
1471 }
1472
```