

```

1  #include "controller.hpp"
2
3  int main() {
4      Controller game;
5      game.startGame();
6  }
7
8
9  /* ----- */
10
11 #ifndef LABO3_CONTROLLER_HPP
12 #define LABO3_CONTROLLER_HPP
13
14 #include <string>
15 #include <list>
16 #include "Containers/container.hpp"
17 #include "Containers/bank.hpp"
18 #include "Characters/person.hpp"
19 #include "Characters/boy.hpp"
20 #include "Characters/girl.hpp"
21 #include "Characters/dependentPerson.hpp"
22 #include "Characters/driver.hpp"
23 #include "Characters/thief.hpp"
24 #include "Containers/boat.hpp"
25
26 /**
27  * Classe représentant le controller du jeu
28  * @author Friedli Jonathan
29  * @author Jaquier Alexandre
30  */
31 class Controller {
32 public:
33     /**
34      * Constructeur par défaut de la classe Controller
35      */
36     Controller();
37
38     /**
39      * Opérateur d'affectation de la classe Controller
40      */
41     void operator=(const Controller&) = delete;
42
43     /**
44      * Impossibilité d'utiliser le constructeur de copie
45      * @param other
46      */
47     Controller(const Controller& other) = delete;
48
49     /**
50      * destructeur de la classe Controller
51      */
52     ~Controller();
53
54     /**
55      * Méthode permettant d'afficher un message d'erreur
56      * @param errorMsg le message d'erreur
57      */
58     static void showError(const std::string &errorMsg);
59
60     /**
61      * Méthode permettant de gérer le début du jeu
62      */
63     void startGame();
64 private:
65
66     /**
67      * Méthode permettant de lancer la boucle de jeu
68      */
69     void nextTurn();

```

```

70
71 /**
72  * Crée le bateau, les deux rives ainsi que tous les personnages du jeu.
73  */
74 void initVariables();
75
76 /**
77  * Méthode finissant un tour de jeu
78  */
79 void endTurn();
80
81 /**
82  * Méthode permettant d'embarquer un personnage sur le bateau
83  * @param p le personnage à embarquer
84  */
85 void embark(const Person *p);
86
87 /**
88  * Méthode permettant de débarquer un personnage du bateau
89  * @param p le personnage à débarquer
90  */
91 void disembark(const Person *p);
92
93 /**
94  * Méthode permettant de déplacer le bateau de rive
95  */
96 void moveBoat();
97
98 /**
99  * Méthode permettant d'afficher l'état du jeu
100 */
101 void display() const;
102
103 /**
104  * Méthode permettant de parser les commandes du joueur
105  * @param input la commande entrée par le joueur
106  */
107 void parseInput(const std::string& input);
108
109 /**
110  * Permet de remettre le jeu dans son état initiale afin de le recommencer
111  */
112 void reset();
113
114 /**
115  * Méthode permettant de récupérer l'input du joueur
116  * @return input du joueur
117  */
118 void userInput();
119
120 /**
121  * Méthode permettant d'afficher le menu
122  */
123 static void showMenu();
124
125 /**
126  * Méthode permettant d'afficher une ligne du menu
127  * @param command nom de la commande
128  * @param info description de la commande
129  */
130 static void printMenuLine(char command, const std::string &info, const
131 std::string &argument = "");
132
133 /**
134  * Méthode permettant de convertir un string en personne
135  * @param s le string à convertir
136  * @return le personnage correspondant au string
137  */
138 const Person *compareStringToPerson(const std::string &s) const;

```

```

139
140 /**
141  * Méthode permettant de changer un personnage de container
142  * @param p personnage à changer de place
143  * @param toAdd lieu ou ajouter le personnage
144  * @param addFrom lieu ou enlever le personnage
145  */
146 void changeLocation(const Person &p, Container &toAdd, Container &addFrom);
147
148 /**
149  * Méthode permettant de savoir si le jeu est terminé
150  * @return vrai si le jeu est terminé
151  */
152 bool endOfGame() const;
153
154
155 int turn;
156 bool gameRunning;
157 Bank *leftBank, *rightBank;
158 Boat *boat;
159 std::list<const Person *> people;
160 static const std::string ERROR_MESSAGE, SEPARATOR;
161 static const char DISPLAY = 'p', EMBARK = 'e', DISEMBARK = 'd', MOVE = 'm' ,
162 RESET = 'r', EXIT = 'q', HELP = 'h';
163 };
164
165
166 #endif //LABO3_CONTROLLER_HPP
167
168 /* ----- */
169 #include "controller.hpp"
170 #include <iostream>
171 #include <iomanip>
172
173 const std::string Controller::ERROR_MESSAGE = "Input invalide ! (Tapez \"h\" pour "
174 "obtenir de l'aide) : ";
175 const std::string Controller::SEPARATOR =
176 "-----";
177
178 Controller::Controller() {
179     initVariables();
180 }
181
182 Controller::~Controller() {
183     delete boat;
184     delete leftBank;
185     delete rightBank;
186     for (const Person *p: people) {
187         delete p;
188     }
189 }
190
191 void Controller::initVariables() {
192     Driver *mother = new Driver("mere");
193     Driver *father = new Driver("pere");
194     Boy *paul = new Boy("paul", *father, *mother);
195     Boy *pierre = new Boy("pierre", *father, *mother);
196     Girl *julie = new Girl("julie", *mother, *father);
197     Girl *jeanne = new Girl("jeanne", *mother, *father);
198     Driver *policeman = new Driver("policier");
199     Thief *thief = new Thief("voleur", *policeman);
200     this->people = {mother, father, paul, pierre, julie, jeanne, policeman, thief};
201     turn = 0;
202     leftBank = new Bank("Gauche");
203     rightBank = new Bank("Droite");
204     boat = new Boat(*leftBank);
205     leftBank->addAll(people);
206     gameRunning = true;
207 }

```

```

208 void Controller::startGame() {
209     showMenu();
210     display();
211     nextTurn();
212 }
213
214 void Controller::showMenu() {
215     std::cout << std::endl;
216     printMenuLine(DISPLAY, "afficher");
217     printMenuLine(EMBARK, "embarquer", " <nom>");
218     printMenuLine(DISEMBARK, "debarquer", " <nom>");
219     printMenuLine(MOVE, "deplacer bateau");
220     printMenuLine(RESET, "reinitialiser");
221     printMenuLine(EXIT, "quitter");
222     printMenuLine(HELP, "menu");
223 }
224
225 void Controller::nextTurn() {
226     while (gameRunning) {
227         userInput();
228     }
229 }
230
231 void Controller::parseInput(const std::string &input){
232     char command;
233     const Person *person = nullptr;
234     if (input.empty()) {
235         showError(ERROR_MESSAGE);
236         return;
237     }
238     if (input.size() > 1 && input[1] == ' ') {
239         person = compareStringToPerson(input.substr(2));
240     }
241     command = input[0];
242     switch (command) {
243         case DISPLAY:
244             display();
245             break;
246         case EMBARK:
247             embark(person);
248             break;
249         case DISEMBARK:
250             disembark(person);
251             break;
252         case MOVE:
253             moveBoat();
254             break;
255         case RESET:
256             reset();
257             break;
258         case EXIT:
259             gameRunning = false;
260             break;
261         case HELP:
262             showMenu();
263             break;
264         default :
265             showError(ERROR_MESSAGE);
266     }
267 }
268
269 void Controller::endTurn() {
270     ++turn;
271     display();
272 }
273
274
275
276

```

```

277 void Controller::display() const {
278     std::cout << std::endl << SEPARATOR << std::endl;
279     leftBank->toStream(std::cout);
280     std::cout << std::endl;
281     boat->toStream(std::cout);
282     std::cout << std::endl;
283     rightBank->toStream(std::cout);
284     std::cout << std::endl << SEPARATOR << std::endl;
285 }
286
287 void Controller::reset() {
288     rightBank->emptyContainer();
289     leftBank->emptyContainer();
290     boat->emptyContainer();
291     for (const Person *p: people) {
292         leftBank->addPerson(*p);
293     }
294     boat->moveBoat(*leftBank);
295     turn = 0;
296 }
297
298 void Controller::userInput(){
299     std::string input;
300     std::cout << turn << ">";
301     getline(std::cin, input);
302     parseInput(input);
303 }
304
305 void Controller::printMenuLine(char command, const std::string &info, const
306 std::string &argument) {
307     std::cout << command << " " << std::setw(8) << std::left << argument <<
308         ": " << info << " " << argument << std::endl;
309 }
310
311 const Person *Controller::compareStringToPerson(const std::string &s) const {
312     for (const Person *p: people) {
313         if (p->getName() == s) {
314             return p;
315         }
316     }
317     return nullptr;
318 }
319
320 void Controller::embark(const Person *p) {
321     if (boat->isFull() || !p) {
322         showError("Error: Bateau est plein ou la personne n'a pas ete trouvee");
323     } else if (boat->isOnBank(*p)) {
324         Bank *bank = boat->isDockedOnthisBank(*leftBank) ? leftBank : rightBank;
325         changeLocation(*p, *boat, *bank);
326     } else {
327         showError("Error: La personne n'est pas sur la rive");
328     }
329 }
330
331 void Controller::disembark(const Person *p) {
332     if (boat->isEmpty()) {
333         showError("Error: Le bateau est deja vide");
334     } else if (boat->isMember(*p)) {
335         Bank *bank = boat->isDockedOnthisBank(*leftBank) ? leftBank : rightBank;
336         changeLocation(*p, *bank, *boat);
337     } else {
338         showError("Error: Personne n'est pas dans le bateau");
339     }
340     endOfGame();
341 }
342
343
344
345

```

```

346 void Controller::changeLocation(const Person &p, Container &toAdd, Container
347 &toRemove) {
348     toAdd.addPerson(p);
349     toRemove.removePerson(p);
350     if (!(toAdd.isContainerSafe() && toRemove.isContainerSafe())) {
351         toAdd.removePerson(p);
352         toRemove.addPerson(p);
353     } else {
354         endTurn();
355     }
356 }
357
358 bool Controller::endOfGame() const {
359     if (boat->isEmpty() && leftBank->isEmpty()) {
360         std::cout << "Vous avez gagne !" << std::endl;
361         return true;
362     }
363     return false;
364 }
365
366 void Controller::showError(const std::string &errorMsg) {
367     std::cout << "### " << errorMsg << std::endl;
368 }
369
370 void Controller::moveBoat() {
371     Bank* bank = boat->isDockedOnthisBank(*leftBank) ? rightBank : leftBank;
372     if(boat->moveBoat(*bank)){
373         endTurn();
374     } else {
375         showError("Il n'y a pas de conducteur dans le bateau");
376     }
377 }
378
379 /* ----- */
380
381 #ifndef LABO3_CONTAINER_HPP
382 #define LABO3_CONTAINER_HPP
383
384 #include <string>
385 #include <algorithm>
386 #include <list>
387
388 class Person;
389
390 /**
391  * Classe représentant un conteneur
392  * @author Friedli Jonathan
393  * @author Jaquier Alexandre
394  */
395 class Container {
396 protected:
397     /**
398      * Constructeur de la classe Container
399      * @param name Nom du conteneur
400      */
401     explicit Container(const std::string &name);
402
403 public:
404     /**
405      * Destructeur de la classe Container
406      */
407     virtual ~Container() = 0;
408
409     /**
410      * Méthode permettant de récupérer le container sous forme affichable
411      * @param os opérateur de flux
412      * @return le container sous forme affichable
413      */
414     virtual std::ostream &toStream(std::ostream &os) const;

```

```

415
416 /**
417  * Méthode permettant de récupérer le nom des personnes du conteneur
418  * @return le nom des personnes du conteneur
419  */
420 std::string getPeopleNames() const;
421
422 /**
423  * Méthode permettant de récupérer le nom du conteneur
424  * @return le nom du conteneur
425  */
426 std::string getName() const;
427
428 /**
429  * Méthode permettant de récupérer la taille
430  * @return la taille
431  */
432 unsigned size() const;
433
434 /**
435  * Méthode permettant de vider le container
436  */
437 void emptyContainer();
438
439 /**
440  * Ajoute un personnage au container
441  * @throw std::runtime_error si le container est plein
442  * @param p personnage à ajouter
443  */
444 void addPerson(const Person &p);
445
446 /**
447  * Ajoute plusieurs personnages au container
448  * @throw std::runtime_error si le container est plein
449  * @param people liste de personnages à ajouter
450  */
451 void addAll(const std::list<const Person *> &people);
452
453 /**
454  * Enlève un personne du container
455  * @throw std::runtime_error si le container est vide
456  * @param p personnage à enlever
457  */
458 void removePerson(const Person &p);
459
460 /**
461  * Méthode permettant de savoir si le container est vide
462  * @return true si le container est vide, false sinon
463  */
464 bool isEmpty() const;
465
466 /**
467  * Méthode permettant de savoir si le container est plein
468  * @return true si le container est plein, false sinon
469  */
470 virtual bool isFull() const;
471
472 /**
473  * Méthode permettant de savoir si un personnage est dans le container
474  * @param p personnage à rechercher
475  * @return true si le personnage est dans le container, false sinon
476  */
477 bool isMember(const Person &p) const;
478
479
480
481
482
483

```

```

484     /**
485      * Méthode permettant de savoir si les personnes dans le container sont toutes
486      * en sécurité
487      * @return true si les personnes sont en sécurité, false sinon
488      */
489     bool isContainerSafe() const;
490
491     /**
492      * Méthode retournant un itérateur constant sur le premier élément de la liste
493      * de personnes se trouvant dans le container
494      * @return un itérateur constant sur le premier élément de la liste
495      */
496     std::list<const Person *>::const_iterator begin() const;
497
498     /**
499      * Méthode retournant un itérateur constant sur le dernier élément de la liste
500      * de personnes se trouvant dans le container
501      * @return un itérateur constant sur le dernier élément de la liste
502      */
503     std::list<const Person *>::const_iterator end() const;
504
505 private:
506     const std::string name;
507     std::list<const Person *> people;
508 };
509
510
511 #endif //LABO3_CONTAINER_HPP
512
513 /* ----- */
514
515 #include <iostream>
516 #include <sstream>
517 #include "container.hpp"
518 #include "../controller.hpp"
519
520 Container::Container(const std::string &name) : name(name) {}
521
522 Container::~Container() = default;
523
524 std::ostream &Container::toStream(std::ostream &os) const {
525     return os << getName() << " : " << getPeopleNames() << " ";
526 }
527
528 std::string Container::getPeopleNames() const {
529     std::stringstream ss;
530     for (auto &person: people) {
531         ss << person->getName() << " ";
532     }
533     return ss.str();
534 }
535
536 std::string Container::getName() const {
537     return name;
538 }
539
540 void Container::emptyContainer() {
541     people.clear();
542 }
543
544 void Container::addPerson(const Person &p) {
545     if (isFull()) {
546         throw std::runtime_error("le container est plein");
547     }
548     people.push_back(&p);
549 }
550
551
552

```



```

553 void Container::removePerson(const Person &p) {
554     if (people.empty()) {
555         throw new std::runtime_error("Le container est vide");
556     }
557     people.remove(&p);
558 }
559
560 bool Container::isMember(const Person &p) const {
561     return std::find(people.begin(), people.end(), &p) != people.end();
562 }
563
564 bool Container::isContainerSafe() const {
565     for (const Person *p: people) {
566         if (!p->isSafe(*this)) {
567             Controller::showError(p->getErrorMessage());
568             return false;
569         }
570     }
571     return true;
572 }
573
574 bool Container::isEmpty() const {
575     return people.empty();
576 }
577
578 bool Container::isFull() const {
579     return false;
580 }
581
582 std::list<const Person *>::const_iterator Container::begin() const {
583     return people.cbegin();
584 }
585
586 std::list<const Person *>::const_iterator Container::end() const {
587     return people.cend();
588 }
589
590 unsigned Container::size() const {
591     return people.size();
592 }
593
594 void Container::addAll(const std::list<const Person *> &peopleToAdd) {
595     for (const Person *p: peopleToAdd) {
596         if (isFull())
597             throw std::runtime_error("le container est plein");
598         this->people.push_back(p);
599     }
600 }
601 /* ----- */
602 #ifndef LABO3_BANK_HPP
603 #define LABO3_BANK_HPP
604
605 #include "container.hpp"
606
607 /**
608  * Classe représentant une rive de la rivière
609  * @author Friedli Jonathan
610  * @author Jaquier Alexandre
611  */
612 class Bank : public Container {
613 public:
614     /**
615      * Constructeur de la classe Bank
616      * @param name nom de la rive
617      * @param people personne se trouvant sur la rive
618      */
619     explicit Bank(const std::string &name);
620 };
621 #endif //LABO3_BANK_HPP

```

```

622
623  /* ----- */
624
625  #include "bank.hpp"
626
627  Bank::Bank(const std::string &name) : Container(name) {}
628
629  /* ----- */
630
631  #ifndef LABO3_BOAT_HPP
632  #define LABO3_BOAT_HPP
633
634  #include "bank.hpp"
635  #include "container.hpp"
636
637  /**
638   * Classe représentant un bateau
639   * @author Friedli Jonathan
640   * @author Jaquier Alexandre
641   */
642  class Boat : public Container {
643  public:
644      /**
645       * Constructeur de la classe Boat
646       * @param current rive sur laquelle le bateau se trouve
647       */
648      explicit Boat(const Bank &current);
649
650      /**
651       * Méthode permettant de déplacer le bateau sur une autre rive
652       * @param bank La nouvelle rive
653       * @return true si le déplacement a réussi, false sinon
654       */
655      bool moveBoat(const Bank &bank);
656
657      /**
658       * Méthode permettant de savoir sur quelle rive le bateau se trouve
659       * @param bank rive à checker
660       * @return true si le bateau se trouve sur la rive, false sinon
661       */
662      bool isDockedOnthisBank(const Bank &bank) const;
663
664      /**
665       * Méthode permettant de récupérer le bateau sous forme affichable
666       * @param os opérateur de sortie
667       * @return le bateau sous forme affichable
668       */
669      std::ostream &toStream(std::ostream &os) const override;
670
671      /**
672       * Méthode permettant de savoir si une personne est dans le bateau
673       * @param person Personne à rechercher
674       * @return true si la personne est dans le bateau, false sinon
675       */
676      bool isOnBank(const Person& person) const;
677
678      /**
679       * Méthode permettant de savoir si le bateau est rempli
680       * @return true si le bateau est rempli, false sinon
681       */
682      bool isFull() const override;
683
684  private:
685      const Bank *currentBank;
686      static const std::string RIVER;
687      static const int MAX_CAPACITY = 2;
688  };
689
690  #endif //LABO3_BOAT_HPP

```

```

691
692  /* ----- */
693
694  #include <iostream>
695  #include "boat.hpp"
696  #include "../controller.hpp"
697
698  const std::string Boat::RIVER =
699      "=====";
700
701  Boat::Boat(const Bank &current) : Container("Bateau"), currentBank(&current) {}
702
703
704  std::ostream &Boat::toStream(std::ostream &os) const {
705      if (currentBank->getName() == "Droite")
706          os << std::endl << RIVER << std::endl;
707
708      os << Container::getName() << " : " << "< " << Container::getPeopleNames() <<
709          ">";
710      if (currentBank->getName() == "Gauche")
711          std::cout << std::endl << RIVER << std::endl;
712      return os;
713  }
714
715  bool Boat::moveBoat(const Bank &bank) {
716      bool hasDriver = false;
717      auto end = Container::end();
718      for(auto it = Container::begin(); it != end; ++it) {
719          if ((*it)->canDrive()) {
720              hasDriver = true;
721              break;
722          }
723      }
724      if (!hasDriver) {
725          return false;
726      }
727      currentBank = &bank;
728      return true;
729  }
730
731  bool Boat::isDockedOnthisBank(const Bank &bank) const {
732      return &bank == currentBank;
733  }
734
735  bool Boat::isFull() const {
736      return Container::size() >= MAX_CAPACITY;
737  }
738
739  bool Boat::isOnBank(const Person &person) const {
740      return currentBank->isMember(person);
741  }
742
743  /* ----- */
744
745  #ifndef LABO3_PERSON_HPP
746  #define LABO3_PERSON_HPP
747
748  #include <list>
749  #include <string>
750
751  class Container;
752  /**
753   * Classe représentant une personne
754   * @author Friedli Jonathan
755   * @author Jaquier Alexandre
756   */
757
758
759

```

```

760 class Person {
761 protected:
762     /**
763      * Constructeur de la classe Person
764      * @param name nom de la personne
765      */
766     explicit Person(const std::string& name);
767
768 public:
769     /**
770      * Destructeur de la classe Person
771      */
772     virtual ~Person() = 0;
773
774     /**
775      * Méthode permettant de savoir si la personne peut conduire
776      * @return true si la personne peut conduire, false sinon
777      */
778     virtual bool canDrive() const;
779
780     /**
781      * Méthode permettant de connaître le nom de la personne
782      * @return le nom de la personne
783      */
784     std::string getName() const;
785
786     /**
787      * Méthode permettant de savoir si la personne est en sécurité dans un lieu
788      * @param people liste des personnes présentes dans le lieu
789      * @return true si la personne est en sécurité, false sinon
790      */
791     virtual bool isSafe(const Container& container) const;
792
793     /**
794      * Méthode permettant d'avoir un message d'erreur lié à la personne
795      * @return un message d'erreur
796      */
797     virtual std::string getErrorMessage() const;
798
799 private:
800     const std::string name;
801     static const std::string ERROR_MESSAGE;
802 };
803
804
805 #endif //LABO3_PERSON_HPP
806
807 /* ----- */
808 #include "person.hpp"
809
810 #include "../Containers/container.hpp"
811
812 const std::string Person::ERROR_MESSAGE = "aucune erreur possible";
813
814 Person::~Person() = default;
815
816 Person::Person(const std::string &name) : name(name) {}
817
818 bool Person::canDrive() const {
819     return false;
820 }
821
822 std::string Person::getName() const {
823     return name;
824 }
825
826 bool Person::isSafe(const Container &container) const {
827     return true;
828 }

```

```

829
830 std::string Person::getErrorMessage() const {
831     return ERROR_MESSAGE;
832 }
833
834
835 /* ----- */
836
837 #ifndef LABO3_DRIVER_HPP
838 #define LABO3_DRIVER_HPP
839
840 #include "person.hpp"
841
842 /**
843  * Classe représentant les conducteurs
844  * @author Friedli Jonathan
845  * @author Jaquier Alexandre
846  */
847 class Driver : public Person {
848 public:
849     /**
850      * Constructeur de la classe Driver
851      * @param name nom du conducteur
852      */
853     explicit Driver(const std::string &name);
854
855     /**
856      * Méthode permettant de savoir si la personne peut conduire
857      * @return true si la personne peut conduire, false sinon
858      */
859     bool canDrive() const override;
860 };
861
862
863 #endif //LABO3_DRIVER_HPP
864
865
866 /* ----- */
867
868 #include "driver.hpp"
869
870 Driver::Driver(const std::string &name) : Person(name) {}
871
872 bool Driver::canDrive() const {
873     return true;
874 }
875
876
877 /* ----- */
878
879 #ifndef LABO3_THIEF_HPP
880 #define LABO3_THIEF_HPP
881
882
883 #include "person.hpp"
884
885 /**
886  * Classe représentant un voleur
887  * @author Friedli Jonathan
888  * @author Jaquier Alexandre
889  */
890 class Thief : public Person {
891 public:
892     /**
893      * Constructeur de la classe Thief
894      * @param name nom du voleur
895      * @param goodWith Personne avec qui le voleur peut rester
896      */
897     Thief(const std::string &name, const Person &goodWith);

```

```

898
899     /**
900     * Méthode permettant de savoir si la personne est en sécurité dans un lieu
901     * @param people liste des personnes présentes dans le lieu
902     * @return true si la personne est en sécurité, false sinon
903     */
904     bool isSafe(const Container &container) const override;
905
906     /**
907     * Méthode permettant d'avoir un message d'erreur lié à la personne
908     * @return un message d'erreur
909     */
910     std::string getErrorMessage() const override;
911
912 private:
913     static const std::string ERROR_MESSAGE;
914     const Person *dependsOn;
915 };
916
917
918 #endif //LABO3_THIEF_HPP
919
920
921 /* ----- */
922
923 #include "thief.hpp"
924 #include "../Containers/container.hpp"
925
926 const std::string Thief::ERROR_MESSAGE = "voleur sans policier";
927
928 Thief::Thief(const std::string &name, const Person &goodWith) : Person(name) {
929     this->dependsOn = &goodWith;
930 }
931
932 bool Thief::isSafe(const Container &container) const {
933     if (container.size() == 1) {
934         return true;
935     }
936     for (auto it = container.begin(); it != container.end(); ++it) {
937         if (*it == this->dependsOn) {
938             return true;
939         }
940     }
941     return false;
942 }
943
944 std::string Thief::getErrorMessage() const {
945     return ERROR_MESSAGE;
946 }
947
948 /* ----- */
949
950 #ifndef LABO3_DEPENDENTPERSON_HPP
951 #define LABO3_DEPENDENTPERSON_HPP
952
953 #include "person.hpp"
954 #include <algorithm>
955
956 /**
957 * Classe représentant une personne dépendante de deux autre
958 * @author Friedli Jonathan
959 * @author Jaquier Alexandre
960 */
961
962
963
964
965
966

```

```

967 class DependentPerson : public Person {
968 protected:
969     /**
970     * Constructeur de la classe DependentPerson
971     * @param name nom de la personne
972     * @param dependsOn personne de qui la personne dépend
973     * @param badWith personne avec laquelle elle ne peut pas rester
974     */
975     DependentPerson(const std::string &name, const Person &dependsOn, const Person &
976     badWith);
977
978 public:
979     /**
980     * Destructeur de la classe DependentPerson
981     */
982     ~DependentPerson() = 0;
983
984     /**
985     * Méthode permettant de savoir si la personne est en sécurité dans un lieu
986     * @param people liste des personnes présentes dans le lieu
987     * @return true si la personne est en sécurité, false sinon
988     */
989     bool isSafe(const Container &container) const override;
990
991 private:
992     const Person *dependsOn;
993     const Person *badWith;
994 };
995
996
997 #endif //LABO3_DEPENDENTPERSON_HPP
998
999
1000 /* ----- */
1001
1002 #include "dependentPerson.hpp"
1003 #include "../Containers/container.hpp"
1004
1005 DependentPerson::~DependentPerson() = default;
1006
1007 DependentPerson::DependentPerson(const std::string &name, const Person &dependsOn,
1008                                 const Person &badWith) : Person(name) {
1009     this->dependsOn = &dependsOn;
1010     this->badWith = &badWith;
1011 }
1012
1013 bool DependentPerson::isSafe(const Container &container) const {
1014     bool isSafe = true;
1015     for (auto it = container.begin(); it != container.end(); ++it) {
1016         if (*it == this->dependsOn) {
1017             isSafe = true;
1018             break;
1019         } else if (*it == this->badWith) {
1020             isSafe = false;
1021         }
1022     }
1023     return isSafe;
1024 }
1025
1026 /* ----- */
1027
1028 #ifndef LABO3_GIRL_HPP
1029 #define LABO3_GIRL_HPP
1030
1031 #include "dependentPerson.hpp"
1032
1033
1034
1035

```

```

1036 /**
1037  * Classe représentant la classe Girl
1038  * @author Friedli Jonathan
1039  * @author Jaquier Alexandre
1040  */
1041 class Girl : public DependentPerson{
1042 public:
1043
1044     /**
1045     * Constructeur de la classe Girl
1046     * @param name nom de la personne
1047     * @param dependsOn personne dont la personne dépend
1048     * @param badWith personne avec laquelle elle ne peut pas rester
1049     */
1050     Girl(const std::string& name,const Person& dependsOn, const Person& badWith);
1051
1052     /**
1053     * Méthode permettant d'avoir un message d'erreur lié à la personne
1054     * @return un message d'erreur
1055     */
1056     std::string getErrorMessage()const override;
1057 private:
1058     static const std::string ERROR_MESSAGE;
1059 };
1060
1061
1062 #endif //LABO3_GIRL_HPP
1063
1064
1065 /* ----- */
1066
1067 #include "girl.hpp"
1068
1069 const std::string Girl::ERROR_MESSAGE = "fille avec son pere sans sa mere";
1070
1071 Girl::Girl(const std::string &name,const Person &dependsOn,const Person &badWith) :
1072 DependentPerson(name,dependsOn,badWith) {}
1073
1074 std::string Girl::getErrorMessage() const {
1075     return ERROR_MESSAGE;
1076 }
1077
1078 /* ----- */
1079
1080 #ifndef LABO3_BOY_HPP
1081 #define LABO3_BOY_HPP
1082
1083 #include "dependentPerson.hpp"
1084
1085 /**
1086  * Classe représentant un garçon
1087  * @author Friedli Jonathan
1088  * @author Jaquier Alexandre
1089  */
1090 class Boy : public DependentPerson {
1091 public:
1092     /**
1093     * Constructeur de la classe Boy
1094     * @param name nom de la personne
1095     * @param dependsOn personne dont il dépend
1096     * @param badWith personne avec qui il ne peut pas rester
1097     */
1098     Boy(const std::string &name, const Person &dependsOn, const Person &badWith);
1099
1100     /**
1101     * Méthode permettant d'avoir un message d'erreur lié à la personne
1102     * @return un message d'erreur
1103     */
1104     std::string getErrorMessage() const override;

```



```
1105
1106 private:
1107     static const std::string ERROR_MESSAGE;
1108 };
1109
1110
1111 #endif //LABO3_BOY_HPP
1112
1113
1114 /* ----- */
1115
1116 #include "boy.hpp"
1117
1118 const std::string Boy::ERROR_MESSAGE = "garcon avec sa mere sans son pere";
1119
1120 Boy::Boy(const std::string &name, const Person &dependsOn, const Person &badWith) :
1121     DependentPerson(name, dependsOn, badWith) {
1122
1123 }
1124
1125 std::string Boy::getErrorMessage() const {
1126     return ERROR_MESSAGE;
1127 }
```