Forestier Quentin & Herzig Melvyn

POO2 – 28.04.2021

Rivière

Labo3



Table des matières

[Introduction 2](#_Toc70535341)

[Implémentation 2](#_Toc70535342)

[Personnes 2](#_Toc70535343)

[IndependentPerson 3](#_Toc70535344)

[DependentPerson 3](#_Toc70535345)

[Considération sur les personnes 4](#_Toc70535346)

[Conteneurs 5](#_Toc70535347)

[Bank 5](#_Toc70535348)

[Boat 5](#_Toc70535349)

[Considération sur les conteneurs 5](#_Toc70535350)

[Contrôleur 6](#_Toc70535351)

[Tests 6](#_Toc70535352)

[Annexes 6](#_Toc70535353)

# Introduction

Ce laboratoire porte sur l’implémentation du jeu de la rivière[[1]](#footnote-1) en mode console.

Plusieurs variantes sont possibles. La notre contient les personnages suivants :

* Le policier.
* La mère.
* Le père.
* Le voleur.
* Les deux garçons (paul et pierre).
* Les deux filles (julie et jeanne).

Les 8 protagonistes sont sur la rive gauche d’une rivière. Ils doivent tous se retrouver du côté droite en utilisant un bateau à 2 places.

Les situations suivantes ne doivent jamais avoir lieu ni dans le bateau, ni sur les rives :

* Le voleur est avec le famille sans le policier.
* Un des garçons est seul avec sa mère et son père n’est pas présent.
* Une des filles est seule avec père et sa mère n’est pas présente.

Les enfants ainsi que le voleur ne peuvent pas piloter le bateau.

# Implémentation

## Personnes

La classe *Person* sert de base à toutes les personnes. Elle fournit le nom de la personne ainsi que trois méthodes abstraites :

*// La personne peut-elle conduire le bateau ?*

bool canDrive() const

*// La personne peut-elle rester avec celles en paramètres ?*

bool checkState(const list<const Person\*>& persons) const

*// Explication des relations de la personne.*

string getErrorMessage() const

Nous avons remarqué deux catégories de personnes :

* Les personnes indépendantes (*IndependentPerson*) qui peuvent conduire et qui n’ont pas de problèmes avec les autres. Cela inclus le policier, la mère et le père.
* Les personnes dépendantes (*DependentPerson*) qui ne peuvent pas rester avec n’importe qui sans leur « *tutor* ». Cela inclus les enfants et le voleur.

### IndependentPerson

Cette classe est une spécialisation de *Person*

Voici le comportement des trois méthodes héritées :

bool canDrive() const :

retourne true.

bool checkState(const list<const Person\*>& persons) const :

retourne true car n’a aucune relation problématique.

string getErrorMessage() const :

retourne un string vide. La personne n’a pas d’explications car pas de relations problématique.

Les personnages concernés héritent directement de cette classe

### DependentPerson

La classe *DependentPerson* offre une méthode pour ajouter un tueur et des personnes problématiques. Sans la présence du tuteur, la personne en question ne peut pas rester seule avec les personnes problématiques.

void setRule(const Person\* tutor, initializer\_list<const Person \*> persons)

Cette opération pourrait être faite via un constructeur. Toutefois, pour laisser plus de liberté dans l’ordre de création des personnages, nous avons décidé que la définition des « relations » serait faite par une méthode.

Cette classe implémente :

bool canDrive() const :

retourne false.

bool checkState(const list<const Person\*>& persons) const :

Vérifie pour chaque personne en paramètre si elle correspond à une personne passée dans la méthode *setRule*. Si c’est le cas, retourne false.

Nous avons ensuite défini deux classes qui permettent de spécifier le messages d’erreur.

**Child :**

À la construction, définit si l’enfant est un garçon ou une fille à l’aide d’un booléen.

string getErrorMessage() const

Si c’est un garçon :  
Retourne : @nom ne peut pas rester avec sa mère sans son père.

Si c’est une fille :  
Retourne : @nom ne peut pas rester avec son père sans sa mère.

**Thief**

string getErrorMessage() const

Le voleur ne peut pas rester avec la famille sans le policier.

### Considération sur les personnes

**Avantages**Cette implémentation est très simple. Les personnes sont soit des personnes indépendantes, sans relation problématique et qui peuvent conduire, soit des personnes dépendantes, dans l’incapacité de conduire, avec des problèmes de relation et un message d’erreur personnalisé.

**Inconvénient**Au contraire, il est difficile de faire évoluer les règles. Comment ferions-nous pour avoir des personnes qui peuvent piloter tout en ayant des relations problématiques ? Une solution consisterait à considérer canDrive et checkState comme des comportements spécialisés de Person. Chaque personnage hériterait des comportements qu’il souhaite implémenter.

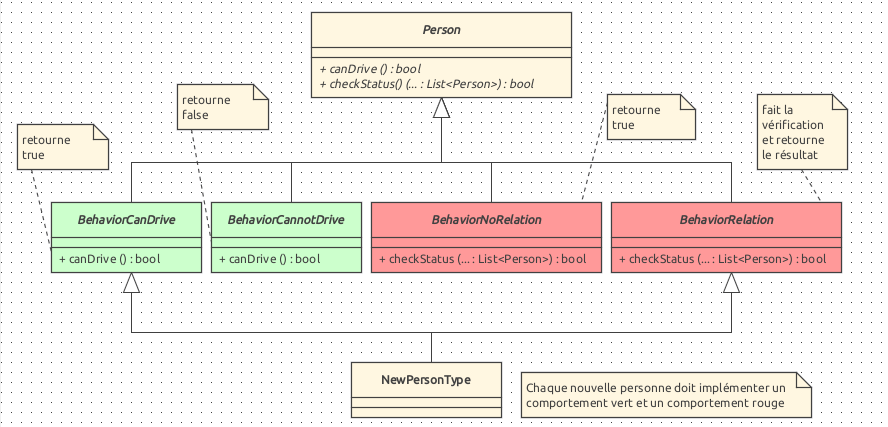


Figure 1 Exemple partiel de l'implémentation des comportements

À l’aide d’héritage virtuel entre *Person* et les *Behavior*, nous serions libres de pouvoir créer des nouveau types de personnes. Cependant cette technique augmente sensiblement le nombre de classe et de spécialisations.

Comme l’évolutivité des « règles » n’était pas un prérequis et que nous ne l’avons pas jugé nécessaire, nous avons implémenté la version simple des personnages.

**Autres remarques**L’implémentation par défaut, du constructeur de copie et de l’opérateur d’affectation, a été laissée. Elles sont suffisamment fonctionnelles et non utiles pour le laboratoire.

## Conteneurs

Les conteneurs modélisent les emplacement des personnes. Cela inclus, le bateau, la rive gauche et la rive droite.

Les conteneurs ont un nom et des personnes sous la forme.

std::list<const Person\*>

Les méthodes principales sont :

*// Ajouter et retirer des personnes, ignore les doubles et les nullptr.*

void addPersons(const std::list<const Person\*>& persons)

void addPerson (const Person\* person);

void removePerson (const Person\* person);

*// Permet de rechercher une personne par son nom*

const Person\* findByName(const std::string& name) const

*// Vérifie si les personnes sont compatibles entre elles, Si elle existe,*

*// retourne la première personne qui n’accepte pas la situation.*

const Person\* isValid() const

*// Définit la manière d’afficher le container dans un flux.*

virtual std::ostream& toStream (std::ostream& os)

### Bank

Cette classe hérite de *Container et* propose sa propre implémentation de toStream

### Boat

Cette classe spécialise *Container* en limitant le nombre de personnes qu’elle peut contenir.

Un bateau navigue entre deux rives :

LEFT\_BANK et RIGHT\_BANK

Il offre de nouvelles méthodes :

*//À tout moment, il est possible de savoir sur quelle rive se situe le bateau.*

const Bank\* getCurrentBank () const;

*//Il est possible de le faire changer de rive :*

void navigate();

*//Vérifie parmi les occupants, si une personne est capable de piloter :*

bool isDriverOnBoard()const

Finalement elle propose sa propre implémentation de toStream.

### Considération sur les conteneurs

Les personnes sont stockées sous la forme de pointeurs constants. Cette manière de faire est légère et permet de retourner les personnes sans risque de les modifier. Deux personnes sont considérées comme égales si elles ont la même adresse mémoire. Dans une version future, il faudrait considérer l’utilisation de *shared\_ptr* car rien ne nous garantit que les personnes soient encore allouées lorsqu’elles sont utilisées.

Dans le cadre de l’application les personnes utilisées sont créées et disparaissent en même temps que les conteneurs. Le risque est donc négligeable. Toutefois pour une utilisation plus générique, cela serait une meilleure pratique.

L’implémentation par défaut, du constructeur de copie et de l’opérateur d’affectation, a été laissée.

## Contrôleur

Cette classe se charge du déroulement de la partie, notamment l’affichage, le traitement et l’exécution des entrées de l’utilisateur.

Cette classe ne peut être créée que par un constructeur sans paramètre qui initialise la partie. Entre autres, il crée les personnes, les rives et le bateau.

La partie peut être lancée au moyen de la méthode :

void play();

La classe alloue dynamiquement les personnes et se charge de les détruire lorsque l’objet cesse d’exister. Pour des raisons de sécurité, le constructeur de copie et d’affectation ont été retirés pour éviter que deux contrôleurs pointent sur les même personnes. Il est envisageable de les rajouter lorsque les personnes seront manipulées au moyen de shared\_ptr.

# Tests

## Tests codés

## Tests utilisation

# Annexes

* Code source (.h et .cpp)
* UML du projet

1. Inspiré de : https://en.wikipedia.org/wiki/River\_crossing\_puzzle [↑](#footnote-ref-1)