HEIG-VD — POA

# Laboratoire 3 – Rapport

[redacted]

23 mai 2024

## 1 Introduction

Dans ce laboratoire, nous allons mettre en pratique différents concepts appris en programmation orientée objet en C++. Pour ce faire, nous avons pour tâche de créer une application interactive basée sur le célèbre jeu de la traversée de la rivière.

Le contexte du problème est le suivant : une famille composée d'un père, d'une mère, de deux fils et de deux filles doit traverser une rivière en compagnie d'un policier et d'un voleur. Plusieurs contraintes doivent être respectées pour garantir la sécurité et l'ordre pendant la traversée. Le bateau ne peut transporter que deux personnes à la fois, seuls les parents et le policier peuvent piloter le bateau, et des règles de surveillance stricte doivent être observées : le voleur ne peut être laissé seul avec un membre de la famille sans la présence du policier, les fils ne peuvent rester seuls avec leur mère sans la présence du père, et les filles ne peuvent rester seules avec leur père sans la présence de la mère.

L'objectif est de modéliser ce problème en utilisant une approche orientée objet et de développer une application console permettant à l'utilisateur de simuler les différentes étapes de la traversée tout en respectant les contraintes imposées.

## 2 Instructions de compilation

Nous avons conservé la structure avec *cmake* pour notre implémentation. Nos versions de la suite de compilation utilisée par *cmake* sont les suivantes :

- **cmake** : 3.29.3
- **ninja** : 1.12.1
- **gcc** : 14.1.1 20240507
- Compilé sur Linux avec le kernel 6.9.1

## 3 Modélisation

La modélisation du problème est donnée en figure 1 en annexe.

## 4 Choix et hypothèses de travail

La modélisation de ce problème a fait l'objet de plusieurs challenges intéressants que nous allons détailler au cas par cas, commençant par le plus important : la gestion des règles principales du jeu.

### 4.1 Modélisation des règles de gestion

Un aspect primordial de la modélisation a été de déterminer comment les règles individuelles des personnes pouvaient être implémentées pour qu'elles soient factorisées et indépendantes de l'état général du jeu (on veut juste contrôler l'état pour la rive ou le bateau dans lequel la personne se trouve).

Nous avons par conséquent décidé de régir cette validation d'état au moyen d'une méthode `validateState` prenant une liste de personnes sous la forme d'une référence constante vers le container où la personne se trouve. Cette méthode est déclarée en *pure virtual* dans la classe abstraite `Person`. En effet, il est supposé que chaque type de personne devra implémenter sa propre gestion (qu'elle soit triviale ou non). Ainsi, chaque type de personne peut définir une façon de répondre à la validation proprement et indépendamment d'autres types de personnes, sans compromettre l'encapsulation des données.

Cette méthode de validation retourne une classe spécifique pour la validation des états, `StateResponse`, décrite plus loin.

Pour les aspects plus triviaux, et comme cela a été suggéré dans la donnée, les personnes peuvent également renseigner si elles peuvent conduire le bateau au moyen du getter `canDrive()` et elles ont toutes un nom qui est renseigné dans le constructeur et accessible de façon constante (avec un `std::string_view` pour s'assurer un traitement optimal ne nécessitant pas de copies du nom). À préciser que les constructeurs de la hiérarchie de personnes ne prennent pas de référence constante vers les strings originaux, mais simplement une valeur par copie qui sera ensuite déplacée dans le champ correspondant.

#### 4.1.1 Enfants

Les garçons et les filles ont tous deux des règles de gestion qui agissent différemment, mais nécessitent d'avoir une référence vers la mère et le père. Nous avons donc une classe `Child` qui permet de regrouper ces deux besoins dans un champ et offre les références constantes vers le père et la mère de façon *protected* via un getter.

Nous n'avons pas factorisé la règle de gestion des enfants (avec la définition d'un parent responsable, par exemple), pour limiter le nombre de modifications qui seraient nécessaires en cas de besoin de modification desdites règles.

#### 4.1.2 Voleur

Une petite spécificité intéressante du voleur est que pour que le jeu soit possible, le voleur doit pouvoir être momentanément placé seul sans la surveillance du policier. Cela a facilement pu être modélisé avec notre implémentation, mais reste toutefois un exercice de pensée quant à la bonne rigueur du voleur de ne pas penser à s'échapper.

### 4.2 Gestion des personnes : containers

Comme justement proposé dans la donnée, nous n'avons pas réinventé la roue, autant pour le choix d'utilisation de la structure de données – `std::list` – que pour la structure de classes. Notre implémentation est donc peu surprenante avec une classe abstraite `Container` qui dirige tout l'aspect d'interfaçage entre la liste sous-jacente en fournissant certaines méthodes utilitaires nécessaires pour la bonne gestion du jeu, ainsi que la structure de base pour l'affichage dans la console.

`Container` définit aussi une première gestion de l'état du jeu, avec la méthode `validatePeople` qui permet dans sa version de base d'appeler la méthode `validateState` de toutes les personnes contenues dans le container avec une référence vers le container lui-même. Cette méthode sera appelée par le contrôleur du jeu pour valider si un déplacement demandé par le joueur est valide avant de le « committer », c'est-à-dire le persister comme état réel.

#### 4.2.1 Rives

L'implémentation de la rive est assez triviale grâce à la factorisation et ne fait que d'exposer un constructeur et définir la façon dont elle doit être affichée dans la console.

### 4.3 Bateau

L'implémentation du bateau est toutefois un peu plus complexe, car le bateau doit aussi permettre de valider le nombre de personnes contenues à l'intérieur et valider indépendamment s'il peut être déplacé.

Ainsi, le bateau ajoute à son constructeur une référence vers une rive qui deviendra la rive active, une méthode qui permet de récupérer ladite rive, une méthode qui permet de la changer et une méthode qui permet de valider que le bateau puisse être déplacé.

Cette dernière va itérer sur les personnes contenues dans la liste sous-jacente et s'assurer qu'une d'entre elles définit bien `canDrive` à *true*.

Le bateau offre en outre la validation des personnes contenues, mais ajoute d'abord une vérification sur le nombre maximal de personnes qui peuvent être contenues dedans. Ce nombre a été défini dans une constante,

car vous conviendrez que cela reste une caractéristique du bateau qui devrait être changée de la même manière que si on venait à changer les règles d'une personne.

### 4.4 Résultats de validation

Comme chaque erreur de validation est accompagnée d'un message d'erreur, nous avons fait le choix d'ajouter une classe `StateResponse` qui permet de convier à l'appelant d'une des méthodes de validation de l'état réel de la validation avec, optionnellement, le message d'erreur en cas de refus.

Cette classe n'offre pas de constructeur public, car cela nous permet de forcer la raison à être définie que au cas où l'état serait invalide au moyen de deux méthodes statiques de construction : `StateResponse::ok` et `StateResponse::nok`. Par conséquent, `ok` ne prend pas de paramètres et retourne directement une instance valide, alors que `nok` va obligatoirement demander la raison en paramètre qui sera ensuite déplacée dans le champ optionnel correspondant.

Nous considérons ainsi un appel au getter `reason` d'être une erreur de logique signalée avec l'exception `logic_error`, c'est donc à l'appelant de faire une gestion intelligente en fonction de l'état retourné.

### 4.5 Coordination du jeu

Le contrôleur, implémenté par la classe `Controller`, va donc coordonner toutes nos classes pour vérifier le bon déroulement du jeu. Nous avons tenu à ce que le contrôleur ne gère en aucun cas les règles de gestion du jeu, sa seule responsabilité doit être de récupérer les inputs de l'utilisateur, vérifier la bonne validité de ceux-ci et gérer l'état global du jeu. Il doit donc utiliser les différentes méthodes fournies par nos classes auxiliaires pour s'assurer que l'état du jeu ne soit à aucun moment compromis, et ainsi permettre un affichage toujours correct dans la console.

Pour ne pas avoir à gérer la mémoire dans le contrôleur, nous prenons en constructeur la liste des personnes qui doivent faire partie du jeu – renforçant par ailleurs la distinction entre la partie « métier » du jeu et la logique d'affichage. Cette liste de personne sera utilisée pour initialiser la liste globale des personnes (gérée par le contrôleur pour la recherche des personnes lors des commandes ainsi que la vérification de la fin de la partie), et l'initialisation de la rive gauche qui contient par défaut toutes les personnes.

Le programme peut ensuite appeler la méthode `run` pour démarrer le jeu, cette méthode est bloquante. En cas de signal nécessitant un arrêt soudain du programme, il est toujours possible d'appeler `stop` qui arrêtera le déroulement de la partie.

À chaque tour, l'utilisateur peut entrer une commande qui est définie dans une `std::map`. Cette map contient pour chaque identifiant de commande une description et une action correspondante à effectuer sur le contrôleur. Elle permet en outre d'être flexible au cas où d'autres commandes serait nécessaire, par exemple, avec l'ajout de méthodes de triches ou d'avancement pour effectuer certains tests.

Une méthode générique `handleCommand` va s'occuper de récupérer la ligne entrée par l'utilisateur, trouver la commande demandée, et appeler son action. Elle affichera un message d'erreur le cas échéant.

Pour factoriser la gestion des déplacements, une méthode `movePerson` a été créée. Cette méthode va déterminer la rive active (celle où se trouve le bateau) et va faire une copie de la rive et du bateau. Cela permet ensuite d'essayer d'applique le mouvement désiré (embarquement ou débarquement) sur les containers copiés. Si le déplacement ne peut pas être effectué, le déplacement est annulé et les copies seront automatiquement détruites. Si le déplacement a pu être appliqué, le contrôleur va ensuite appeler les méthodes `validatePeople` de la rive et du bateau pour vérifier l'état des règles de gestion.

Si l'état retourné est valide, alors les copies seront déplacées dans la rive et le bateau « réels », persistant ainsi le déplacement demandé par l'utilisateur et s'assurant qu'aucun mouvement invalide n'a pu être persisté dans l'état réel du contrôleur.

Une validation similaire est effectuée avant de déplacer le bateau, permettant de d'abord vérifier qu'il peut être déplacé avant d'inverser la banque active du jeu.

Quand toutes les personnes du jeu se trouvent dans la rive droite, le jeu est terminé. Cette vérification est faite après chaque tour, affiche un message en cas de succès et arrête la boucle principale du jeu.

## 5 Tests effectués

Nous avons fait une première approche de tests unitaires dans une branche disponible à part. Ces tests ont été effectués en capturant la sortie standard, une technique qui nous permet d'intercepter et d'analyser les retours console générés par les commandes exécutées. Cette méthode est mise en œuvre par la redirection du buffer de sortie standard vers un flux de sortie tel que `std::ostringstream`. Après l'exécution des commandes, la sortie est récupérée à partir de ce flux, permettant une vérification des résultats attendus, tels que la présence de termes spécifiques indiquant le déroulement des opérations. Toutefois, nous avons décidé de ne pas implémenter complètement cette approche car elle impliquait quelques modifications au code existant.

La plupart des tests ont donc été effectués manuellement à travers la console, permettant de vérifier le bon fonctionnement de l'application en conditions réelles et d'identifier d'éventuels bugs ou comportements inattendus. Cette combinaison de tests unitaires et de tests manuels nous a permis de nous assurer de la robustesse et de la fiabilité de notre solution.

| **Cas de test** | **Cas évalués** | **Résultat attendu** |
|---|---|---|
| TestBoyValidation | Validation du garçon sans son père | Le test doit retourner invalide (false) |
| TestGirlValidation | Validation de la fille sans sa mère | Le test doit retourner invalide (false) |
| TestThiefValidation | Validation du voleur sans le policier | Le test doit retourner invalide (false) |
| TestBoyOtherFatherValidation | Validation du garçon avec un faux père | Le test doit retourner invalide (false) |
| TestGirlOtherMotherValidation | Validation de la fille avec une fausse mère | Le test doit retourner invalide (false) |
| OptimalSolution | Exécution d'une série de commandes donnant la solution optimale | "Bravo, la partie est reussie!" doit apparaître dans la sortie |
| Solution | Exécution d'une série de commandes donnant une solution | "Bravo, la partie est reussie!" doit apparaître dans la sortie |
| TestBadCommand | Tentative d'exécution d'une commande invalide | "Commande invalide" doit apparaître dans la sortie |
| TestBadMoveVoleurSeul | Déplacement du voleur seul sans conducteur | "le bateau n'a pas de conducteur" doit apparaître dans la sortie |
| TestGarconSansPereAvecMere | Garçon avec sa mère sans son père | "garcon avec sa mere sans son pere" doit apparaître dans la sortie |
| TestFilleSansMereAvecPere | Fille avec son père sans sa mère | "fille avec son pere sans sa mere" doit apparaître dans la sortie |
| TestPolicierSansVoleur | Policier sans le voleur | "voleur sans le policier" doit apparaître dans la sortie |
| TestBarqueTropChargee | Surcharge de la barque | "le bateau a trop de personnes" doit apparaître dans la sortie |
| TestVideMove | Déplacement du bateau sans conducteur | "le bateau n'a pas de conducteur" doit apparaître dans la sortie |
| TestPersonneIntrouvable | Embarquement d'une personne non existante | "Personne introuvable" doit apparaître dans la sortie |
| TestDebarquementIntrouvable | Débarquement d'une personne non présente sur le bateau | "La personne n'est pas sur le bateau" doit apparaître dans la sortie |
| TestReset | Réinitialisation du jeu | Descriptions d'état initial correctes après reset doivent apparaître dans la sortie |

Table 1 – Cas de tests et résultats attendus pour RiverTest

## 6 Conclusion

Ce laboratoire nous a permis de mettre en pratique de nombreux concepts avancés de la programmation orientée objet en C++. Nous avons utilisé des pointeurs intelligents pour gérer la mémoire de manière sûre et efficace, en évitant les fuites de mémoire. L'héritage a été employé pour structurer notre hiérarchie de classes, permettant une conception modulaire et extensible. Nous avons également utilisé des fonctions anonymes pour simplifier la gestion des commandes, rendant notre code facilement maintenable. Les sémantiques de déplacement ont été appliquées pour optimiser la gestion des ressources, minimisant les copies inutiles et améliorant les performances. En conclusion, ce laboratoire nous a offert une excellente opportunité d'appliquer les nouvelles connaissances en C++ et en programmation orientée objet, tout en résolvant un problème si emblématique.

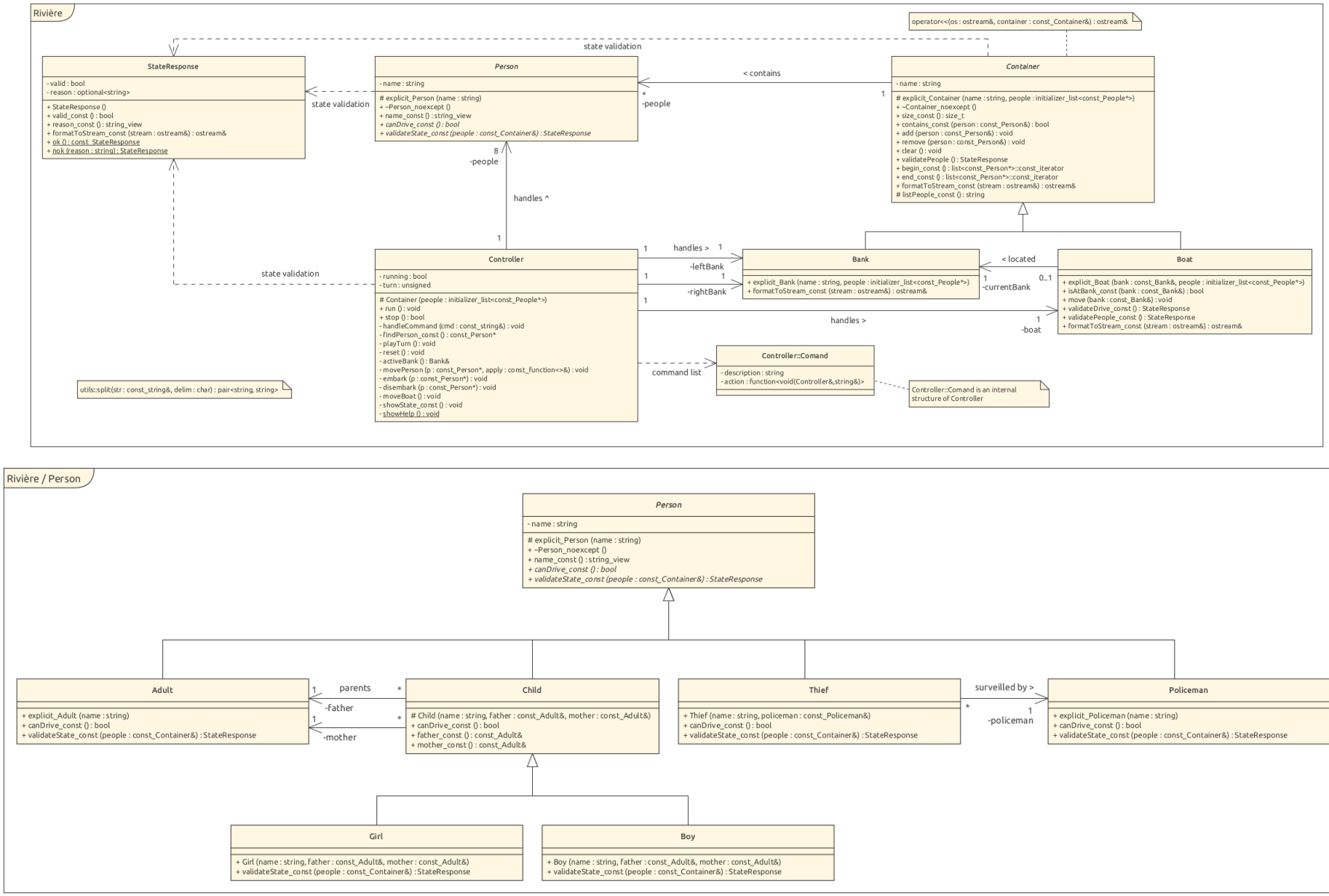


FIGURE 1 – Diagrammes de classe de l'implémentation du problème