Laboratoire 16 : Rivière

## Description de la problématique :

## Une famille composée d'un père, d’une mère, de deux filles et de deux garçons est accompagnée d'un policier et d'un voleur menotté. Ils doivent tous traverser une rivière à l'aide d'un bateau.

## Contraintes devant être satisfaites en tout temps sur les deux rives et sur le bateau :

## • Il ne peut y avoir que deux personnes au maximum sur le bateau.

## • Les enfants et le voleur ne peuvent piloter le bateau (mais peuvent y monter lorsqu’il est à quai).

## • Le voleur ne peut pas rester en contact avec un membre de la famille si le policier n'est pas présent.

## • Les fils ne peuvent rester seuls avec leur mère si le père n’est pas présent.

## • Les filles ne peuvent rester seules avec leur père si la mère n’est pas présente.

## Le but du laboratoire est de créer une application en C++ en mode console permettant à l’utilisateur d’introduire les commandes pour embarquer et débarquer des personnes et déplacer le bateau.

## Architecture :

Nous avons décidé de décomposer le problème en une architecture MVC. Un modèle contient l’ensemble des données physiques (personnes et container), un Controller gère les commandes et s’assure que les règles établies sont bien respectées et une vue se charge de display le tout de manière formatée.

## Le modèle :

Nous avons décidé de mettre la plupart des données utiles directement dans les super-classes et de faire une redéfinition si nécessaire dans les sous-classes. Nous pouvons donc facilement ajouter des classes en redéfinir les propriétés nécessaires pour assurer le respect des règles concernant ce nouveau type.

### Les personnes

Pour représenter les différents types de personnes, nous avons créé dans un premier temps une classe Personne pour englober l’ensemble des membres et factoriser un maximum le code en commun.

La plupart des attributs sont initialisés par défaut afin de permettre de rapidement ajouter une classe sans se soucier de devoir définir un certain nombre de propriétés non utile pour ce type. Une personne est constituée de :

- Un nom pour savoir à qui on a affaire

- Un genre pour appliquer les règles fils/filles sans leur père/mère

- Un booléan pour indiquer si la personne peut conduire

Cette classe Personne se divise en trois sous-classes, les membres de la famille, le policier et le voleur. Ces classes permettant une meilleure représentation des données et redéfinissent les attributs en fonction de leurs besoins (les enfants ne peuvent pas conduire par exemple).

Nous avons utilisé des move pour les noms afin d’éviter de recopier plusieurs fois des données lourdes comme les string.

### Les containers

Nous avons fait une classe commune pour représenter les containers. Ces derniers sont composés de :

* Un nom pour connaître à quel container nous nous référons (par exemple gauche ou droite)
* Une liste représentant l’ensemble des personnes contenues dans l’instance en question
* Un nombre maximum de personnes pouvant être contenues.

Ces containers sont redéfinis en deux sous-classes, Bank pour les berges et Boat pour le bateau. Les berges n’ont pas de limitations du nombre de personnes, par contre le bateau ne peut contenir que deux personnes au maximum en même temps.

De plus, le bateau contient un pointeur sur la berge à laquelle il est arrimé afin de pouvoir embarquer/débarquer des personnes rapidement.

## La vue :

La vue est relativement simple, sachant que le display se fait en console. Nous avons défini des constantes globales pour l’affichage des lignes ainsi que celui du menu. La vue ne possède qu’un pointeur sur le modèle pour connaître son état courant.

Nous avons également défini un certain nombre de fonctions pour avoir un affichage de chaque composant et permettre une meilleure réutilisabilité. Ainsi, il est possible d’afficher uniquement le menu, uniquement une des berges, uniquement la rivière, etc…

## Le Controller :

### RuleFunctionDef

Nous avons déjà dans un premier temps défini un nouveau typedef. Ce dernier comporte :

* Si nécessaire, le container sur lequel on se focalise
* Si nécessaire, la commande entrée si celle-ci se rapporte à une personne, ce pour savoir sur laquelle on tente d’appliquer une règle

La définition de ce nouveau type permet de rapidement chercher les informations utiles pour cette règle et nous permet d’avoir quelque chose de plus standardisé.

### Les règles

Une règle est définie par :

* Le char représentant la commande qui entraîne sa vérification
* Un booléan indiquant si la règle doit être testée avant ou après l’application de la commande
* Un booléan indiquant à quel type de container se rapporte la règle (bateau ou berge)
* La fonction à tester pour savoir si la règle est respectée ou non
* Le message à afficher en cas d’erreur

Les règles sont définies par la fonction setRules du Controller, ceci nous permettant de rapidement en ajouter une nouvelle à tester.

### CommandMethodDef

Nous avons défini un nouveau typedef. Ce dernier représente la fonction à utiliser pour l’application d’une commande.

### Les commandes

Les commandes sont définies sous la forme d’une structure comportant les champs suivants :

* Le char utilisé pour appeler la commande
* La fonction à appliquer lors de l’appel à cette commande si il n’y a pas d’erreur de précondition
* La fonction à appliquer lors de l’appel à cette commande pour un retour à l’état initial si il y a une erreur de postcondition après l’application
* Le booléan indiquant si il faut incrémenter le nombre de tours ou non après l’application de la commande

Les commandes sont définies par la fonction setCommands du Controller, ceci nous permettant de rapidement en ajouter une nouvelle.

### Le Controller

Le Controller va recevoir les commandes, se charger de mettre à jour le modèle en fonction des commandes reçues, s’assurer que l’ensemble des règles sont respectées ainsi que de faire appel à la vue pour mettre à jour l’affichage.

Il contient donc :

* Le nombre de tours auquel est l’utilisateur
* Un pointeur sur le modèle pour modifier ses données
* Un pointeur sur la vue pour display le nouvel état du modèle
* Une liste de commandes pour connaître les commandes autorisées
* Une liste de règles pour connaître les règles à faire respecter
* Une string représentant la dernière commande entrée pour un possible retour en arrière, ce si la commande était valide mais entrainait un non-respect d’une règle
* Un booléan pour savoir si il faut quitter le programme
* Un booléan pour indiquer si il y a une erreur (non-respect d’une règle)
* Une string pour le message à afficher

## Choix architecturaux :

Nous avons choisi de définir une classe pour les règles et une structure pour les commandes. Cette solution a été choisie car les commandes ne possèdent pas vraiment de méthodes. Une structure paraissait de ce fait plus logique. De plus, les commandes sont intrinsèquement liées au Controller.

Les règles, par contre disposent d’un certain nombre de méthodes utiles à leur application et cette classe pourrait potentiellement être utilisée dans un autre projet, moyennant quelques retouches. Mais surtout et avant tout, les règles sont une entité indépendante du Controller.