ESILV LÉONARD DE VINCI | ÉCOLE D'INGÉNIEURS PARIS-LA DÉFENSE

Programmation Orientée Objet et Interface

PFR : Rapport Pour Le Second Rendu

Nicolas Picard
Martin Pintiau

Avril 2018

Contents

1 Modélisation du problème **2**
1.1 Implémentation générale des classes . . . 2
1.2 La place centrale de la classe Administration . . . 2
1.3 Librairies et packages utilisés . . . 2

2 Méthodes implémentées **3**
2.1 Les méthodes informatives et de mise en forme de la donnée . . . 3
2.1.1 Les Constructeurs . . . 3
2.1.2 Les Accesseurs . . . 3
2.1.3 Les méthodes ToString . . . 3
2.1.4 L'interface IExportable . . . 3
2.1.5 Les méthodes de tri . . . 3
2.2 Les méthodes de manipulation de la donnée . . . 3
2.2.1 Lecture et écriture dans un fichier CSV . . . 3
2.2.2 Les méthodes de recherche d'élément . . . 4
2.2.3 Les méthodes de modification du personnel et des attractions . . . 4
2.2.4 Méthodes de modification des attributs du personnel et des attractions . . . 4
2.2.5 Les méthodes de filtrages . . . 4
2.2.6 Les méthodes de tris . . . 5

3 Explications de la démonstration **5**

4 Outils utilisés dans ce projet **5**

5 Annexes **6**
5.1 Diagramme des classes . . . 6
5.2 Répartition du travail dans le temps sur la base des commits réalisés sur git . . . 7

1 Modélisation du problème

1.1 Implémentation générale des classes

Tout d'abord, nous avons choisi de respecter le plus possible la modélisation fournie sur Moodle (voir diagramme des classes en annexe) à l'exception de l'attribut "affectationAutre" que l'on trouve dans la classe "Monstre". Suite à la discussion que nous avons pu avoir avec Mme Branchet sur Yammer, cette chaîne de caractère nous permet de gérer les affectations de type "néant" (*ie. le monstre ne peut pas être affecté à une attraction*) et "parc" tout en conservant un champ "affectation", de type attraction, pour l'attraction affectée aux membres du personnel qui ne fait pas partie de la direction et qui n'est pas directement affecté au parc de manière générale. Ainsi nous pouvons affecter du personnel au parc sans pour autant devoir créer une attraction "parc" qui aurait peu de sens.

1.2 La place centrale de la classe Administration

La gestion se fait donc depuis la classe "Administration" qui ne possède qu'une seule instance par exécution. Cette classe possède également la majeure partie des méthodes qui correspondent aux fonctionnalités du logiciel.

En effet, les seules méthodes que l'on trouvera dans les autres classes sont les constructeurs, les accesseurs (get/set), les méthodes de type ToString servant à décrire l'objet courant, les méthodes qui créent une ligne en format CSV décrivant l'objet courant (pour l'export dans un fichier .csv) et enfin, les méthodes de tri des objets, permettant de répondre aux divers besoins de l'utilisateur.

Au delà de ces méthodes "informatives", toutes les méthodes de gestion appartiennent à la classe Administration. Cela semble plus logique car c'est notre instance de la classe Administration qui possède une liste du personnel et une liste des attractions, qui sont finalement les seuls éléments à gérer dans ce problème. Toutes les opérations à effectuer dans ce logiciel de gestion sont des actions qui, en réalité, sont effectuées par un membre de l'administration.

C'est donc aussi depuis la représentation virtuelle de l'administration que ces actions sont implémentées. Mêmes si certaines de ces fonctionnalités ont vocation à être automatisées, pour être cohérentes vis-à-vis des niveaux d'accessibilités des attributs et des méthodes (à l'aide d'une interface de connexion et d'un profil utilisateur par exemple), il reste impératif de gérer ces fonctionnalités depuis la classe administration.

1.3 Librairies et packages utilisés

Pour résoudre ce problème, notre programme utilise les packages classiques d'une application console en C#. Nous avons également ajouté le "using System.Collections.Generic" pour plus de fonctionnalités dans la manipulation des listes, ainsi que le "using System.Linq" qui nous donne accès aux requêtes LINQ. La totalité des requêtes LINQ pourrait être recodées en C# mais cela impose un code lourd et peu explicite, ce qui n'est pas le cas des commandes LINQ. Cela nous a permis de faire des requêtes semblable au SELECT/FROM/WHERE du SQL mais directement dans nos listes. Nous aurions également pu implémenter directement une base de données SQL pilotée par des requêtes SQL mais nous avons considéré que cela n'était pas assez intéressant vu que nous n'aurions eu que les deux tables personnel et attraction. Nous avons donc juger le C# amplement suffisant.

2 Méthodes implémentées

2.1 Les méthodes informatives et de mise en forme de la donnée

Comme évoqué précédemment, les méthodes qui modifient le personnel et/ou les attractions sont exclusivement incluses dans la classe Administration mais les classes héritières des classes Personnel et Attraction possèdent tout de même certaine méthodes qui servent au fonctionnement du logiciel ainsi qu'à rendre compte de l'état du parc.

2.1.1 Les Constructeurs

Chacune des classes possède son constructeur permettant d'instancier des objets en renseignant chacun de leurs champs ainsi que ceux des classes dont ils héritent. Les données passées en paramètre de ces constructeurs viennent soit de la lecture d'un fichier .csv, soit de la saisie manuelle et individuelle d'un nouvel élément par l'administration. Certain constructeurs possèdent également des automatismes, *(ie. le constructeur de la classe monstre)*. En effet, si le monstre à créer possède une affectation qui existe, on va directement ajouter ce monstre dans le champs "equipe" de l'instance représentant l'attraction en question. De plus, nous profitons aussi des constructeurs pour interpréter les string qui correspondent à des énums. Par exemple, à la création d'un zombie bleuâtre depuis un fichier .csv , nous allons récupérer la chaîne de caractère "bleuâtre", mais le champ couleur de la classe zombie est de type CouleurZ. C'est donc ici que nous l'interprétons via un switch.

2.1.2 Les Accesseurs

Étant donné les niveaux d'accessibilité très restreints que nous avons imposés à la plupart des attributs, nous avons dû implémenter des getters et des setters pour tous ceux susceptibles d'être changés et/ou demandés.

2.1.3 Les méthodes ToString

Pour chacune des classes nous avons implémenté une surcharge de la fonction ToString qui a pour seul but de renvoyer une chaîne de caractères, en français correct, qui décrit avec exactitude l'objet courant. Ces méthodes ne servent qu'à faciliter la compréhension du logiciel par l'utilisateur.

2.1.4 L'interface IExportable

Dans le même genre que les méthodes ToString, chaque classe exportable possède également une méthode ***GetCSVLine*** de l'interface ***IExportable*** qui retourne sous la forme d'une chaîne de caractère la ligne correspondant à la représentation de l'objet dans un format .csv.

2.1.5 Les méthodes de tri

Chaque classe possède différentes méthodes pour comparer ses instances les unes aux autres et trier plusieurs instances d'une liste selon différents paramètres, grâce à l'implémentation de l'interface ***IComparable***.

2.2 Les méthodes de manipulation de la donnée

Comme évoqué précédemment, l'intégralité des méthodes qui interviennent sur les données et les modifient sont implémentées dans la classe Administration.

2.2.1 Lecture et écriture dans un fichier CSV

Après avoir modélisé chacune des classes, les premières fonctions que nous avons implémentées sont celles qui servent à parcourir un fichier .csv en créant chaque instance de chaque objet listé pour représenter le contenu du fichier dans notre modélisation. Nous avons décidé de procéder a deux lectures du fichier qui sont exécutées par les fonctions ***ReadAttractionsFromCSV*** et ***ReadEmployesFromCSV***. La première va d'abord extraire les attractions du fichier et la seconde le personnel, ainsi, quelque soit l'ordre des lignes dans notre fichier, a chaque fois que l'on rencontrera un personnel affecté à une attraction, il sera possible d'ajouter l'attraction dans son champ affectation puisque toutes les attractions du fichier auront été ajoutées auparavant. Dans les deux cas, on lit le fichier ligne par ligne, chaque

champ est ensuite converti au format attendu par les constructeurs de la classe correspondante. Ces constructeurs sont enfin appelés avec les différents champs du fichier dans les bons paramètres du constructeur. L'instance ainsi créee est directement ajoutée dans une des listes d'attractions ou de personnel de l'administration.

Pour l'écriture c'est le même procédé exécuté dans le sens inverse. La fonction ***ExtractListToCSV*** crée le fichier .csv dont le nom est donné en paramètre de la fonction. Puis, également placé en paramètre, elle va parcourir une liste de personnel puis une liste d'attraction. A chaque élément des deux listes la fonction va appeler la fonction de l'interface IExportable (détaillée plus tôt dans le 2.1.4) pour obtenir les lignes au format .csv de chaque éléments et les écrire dans le fichier .csv.

2.2.2 Les méthodes de recherche d'élément

Pour faciliter la recherche de personnel et d'attraction dans nos données, notamment pour savoir si un élément existe avant de le créer une deuxième fois, nous avons implémenté les fonctions ***DoesAttractionExists*** et ***DoesEmployeExists*** . Ces deux fonctions recherchent dans les listes du personnel et des attractions de l'administration des éléments selon les matricules ou l'identifiant qui sont des entiers fournis en paramètres. Si la recherche est positive, les fonctions retournent l'employé ou l'attraction dont il est question.

2.2.3 Les méthodes de modification du personnel et des attractions

Dans cette section, nous avons tout d'abord les fonctions ***AjouterAttraction*** et ***Recruter*** qui vont simplement ajouter aux listes Attractions et toutLePersonnel l'Attraction ou le Personnel passé en paramètre.Ces deux fonctions ont leur réciproque qui sont respectivement ***SupprimerAttraction*** et ***Licencier*** qui ajoutent ou suppriment l'élément passé en paramètre. Nous avons également implémenté les versions plurielles de ces deux fonctions de suppression qui peuvent directement supprimer des listes de personnel ou d'attraction. Cela nous permet par exemple, grâce aux fonctions de filtrage que nous aborderons plus tard, de supprimer un groupe de personnel en fonction d'une caractéristique (tous les zombies ou tous les employés avec une cagnotte inférieure a un certain montant, etc.).
De plus, nous avons implémenté dans cette section la fonction ***CheckCagnotte*** qui va vérifier si la cagnotte d'un monstre passe en dessous de 50 ou au dessus de 500 pour pouvoir appeler si nécessaire, les fonctions qui appliqueront les changements demandés par le sujet, comme par exemple envoyer le monstre dans un stand de Barbe à Papa. Cette fonction est appelée a chaque fois que la cagnotte d'un monstre est modifiée.

2.2.4 Méthodes de modification des attributs du personnel et des attractions

Pour pouvoir maîtriser chaque paramètre de chaque donnée, nous avons créé une fonction de modification pour chaque attribut (du personnel et des attractions) susceptible d'être modifié. On a donc une méthode pour tous les attributs sauf les noms, prénoms, identifiants et matricules qui sont constants. Chacune de ces méthodes prends en paramètre l'identifiant / le matricule de l'objet à modifier ainsi que la modification à lui apporter. Ces fonctions vont d'abord rechercher l'objet en question et, s'il existe, effectuera la modification sur l'attribut en vérifiant les éventuelles contraintes imposées à cet attribut. Afin de répondre à ces contraintes et dans une volonté de toujours factoriser le code au maximum, nous avons implémenté une fonction pour chacune d'entre elles. Le meilleur exemple est la fonction ***SendToBAP*** (BAP = Barbe à Papa) que l'on appelle lorsque la cagnotte d'un monstre est inférieure à 50 et qui affecte notre monstre a un stand à barbe à papa si cela est possible.

2.2.5 Les méthodes de filtrages

Les méthodes de filtrage nous servent à construire des listes de personnel ou d'attraction possédant des caractéristique communes. Cela ne sert pas a la modélisation de la situation du parc mais c'est un outil indispensable pour faciliter le travail du gestionnaire qui utilise le programme.

Tout d'abord, il y a les fonctions de la forme ***Get*****InParc*** et dont il existe deux versions : pour chaque classe héritant de la classe Personnel, on aura des fonctions retournant tous les matricules correspondant à ces types. Pour chaque classe héritant de la classe Attraction, nous avons réalisé le même type de fonction mais récupérant directement les objets en question. Ces fonctions parcourent les listes toutLePersonnel / attractions de l'administration et retournent une liste d'entier ou une liste d'objets

contenant l'ensemble des attractions ou matricules correspondants. Par exemple, la fonction ***GetBoutiquesInParc*** retournera toutes les Boutiques du parc, tandis que ***GetZombieInParc*** retournera la liste des matricules de zombies du parc. Ce sont deux approches différentes car nous avons partagé le travail de cette façon, cependant les deux sont parfaitement fonctionnelles. Ensuite nous avons de nombreuses fonctions pour filtrer selon un attribut, leur fonctionnement et toujours basé sur le même principe. Chaque fonction filtre en fonction d'un seul attribut, la condition que doit respecter cet attribut est fournie en paramètre de la fonction et on fournit également une liste d'objets à filtrer. La liste d'objet à filtrer en paramètre n'est donc pas forcément la liste du personnel ou d'attractions du parc, cela nous permet par exemple de filtrer des listes qui ont déjà été préfiltrées auparavant selon un autre paramètre. Ainsi on peut filtrer par exemple tous les Démons du parc puis à nouveau les filtrer pour obtenir seulement les démons dont la cagnotte est comprise entre 50 et 500. Tous les filtres applicables à un même type d'objet sont alors superposables.

Les paramètres que l'on peut filtrer sont:

- Pour le personnel :
  - le type
  - la cagnotte
  - la fonction
- Pour les attractions :
  - pour chaque attraction le type, le nombre de monstres ou le besoin spécifique
  - pour les boutiques le type de boutique
  - pour les darkrides le type de véhicule, la durée
  - pour les rollercoasters la catégorie, la taille et l'âge minimum
  - pour les spectacles le nombre de places et les noms de salles

2.2.6 Les méthodes de tris

Nous avons implémenté autant de méthodes de tri que de méthodes de filtrage. En effet pour chaque caractéristique des objets que l'on peut filtrer, nous avons implémenté la méthode qui permet d'ordonner une liste d'objet par ordre croissant en fonction de cette caractéristique. Le paramètre discriminant et la liste d'objets sont passés en paramètre et la fonction retourne cette même liste mais ordonnée en fonction du paramètre, en comparant les éléments via les méthodes de tri codées dans chaque classe et que nous avons vu dans la partie 2.1.5.

3 Explications de la démonstration

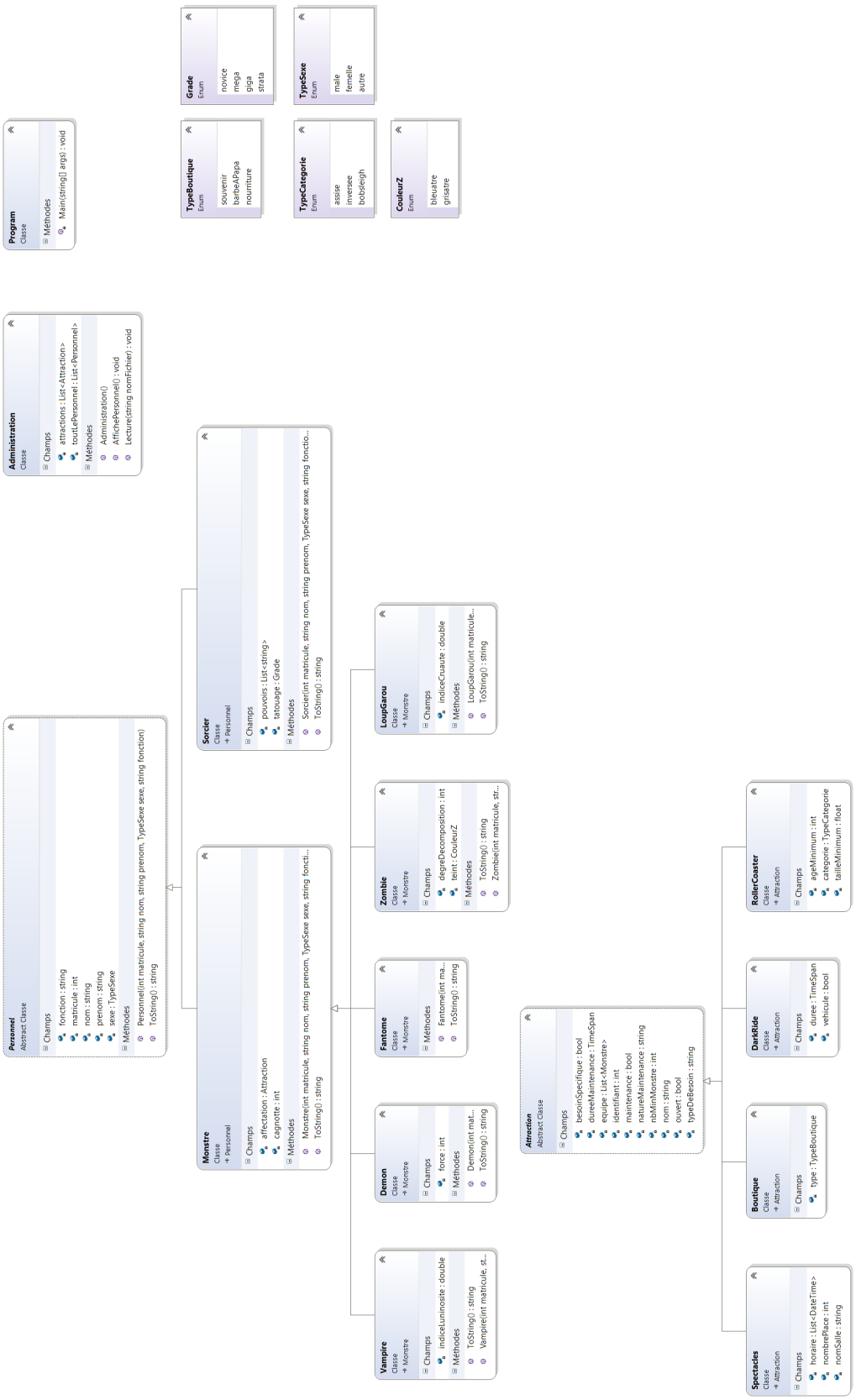
Dans notre démonstration, nous commençons par charger le fichier .csv fourni pour ensuite tester chaque type de fonction que nous avons implémenté. Certaines de nos fonction ne sont pas testées car nous avons pensé qu'il était inutile de tester toutes les fonctions du même type. Par exemple nous n'avons pas fait la démonstration de toutes les fonctions qui modifient les attributs du personnel, nous avons pensé que quelque exemples étaient largement suffisants. C'est aussi le cas des différents filtres et tris, si nous les avions tous ajoutés, la démonstration aurait été beaucoup trop longue. Cependant, ceux-ci sont toujours possibles par l'ajout de quelques lignes.

4 Outils utilisés dans ce projet

Dans ce projet, nous avons codé en langage c# avec les librairies standards d'une application console, dont le package Collection pour ses fonctionnalités sur les listes et le package LINQ pour ses requêtes. Tout le développement s'est fait via l'IDE Visual Studio Community avec les version 2015 et 2017. Nous avons également utilisé Git via GitHub pour l'hébergement du code, le gestionnaire de version et la simplicité du partage des productions. Enfin, le présent rapport est rédigé en LaTeXsur le site d'édition de texte collaboratif Overleaf.

5 Annexes

5.1 Diagramme des classes



5.2 Répartition du travail dans le temps sur la base des commits réalisés sur git