Mini Projet :

Patrons de conception

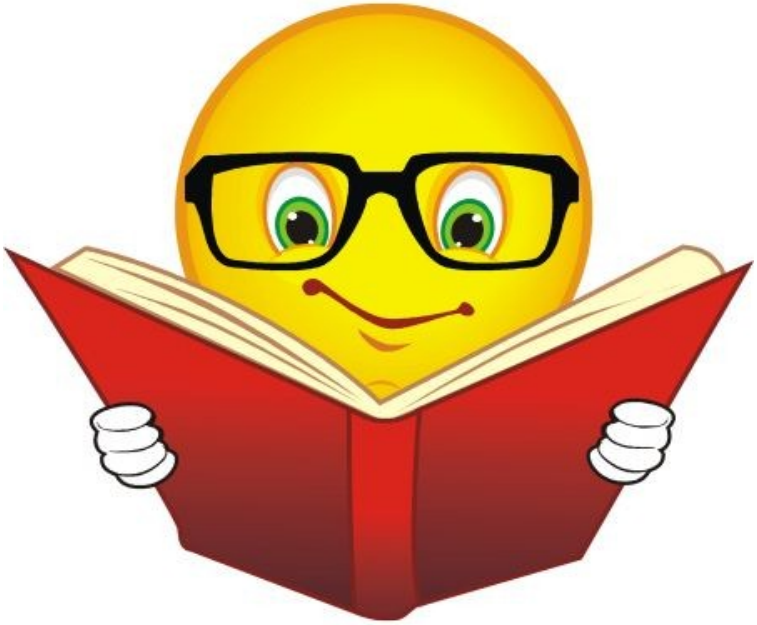
BEUGNET Baptiste, CAILLAUD Antoine, CAILLAULT Mallory, CROAIN Marc-Eli

FLORENT Thomas, ROUSSEL Gauthier

Ce document est composé de 2 parties :

- Dans la 1ère, nous répondrons aux points énoncés dans la consigne quant au choix des pattern, avec illustrations UML.

- Dans la 2nde, nous évoquerons les sujets qui ont fait débats lors de notre réflexion.

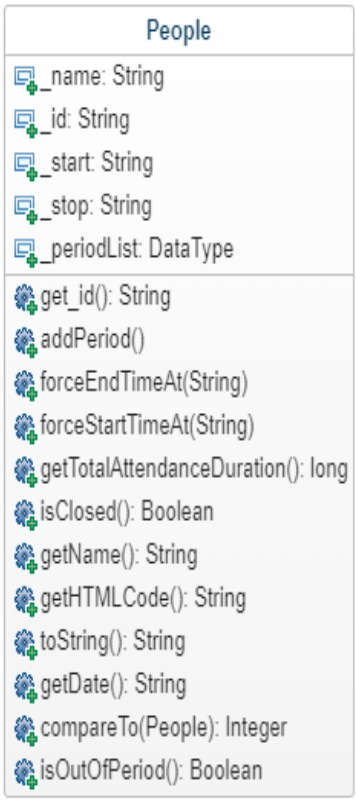


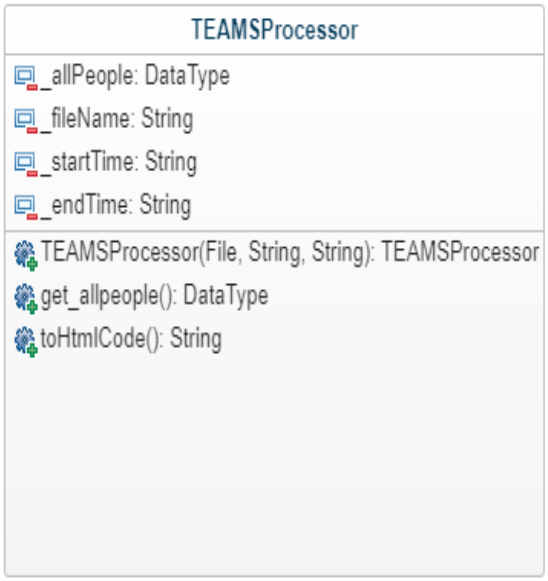
Bonne lecture :)

I / Choix du patron adapté à chaque consigne

1. Factory :

L'application doit pouvoir, à partir d'un objet, générer différents type de fichiers, soit ici HTML ou CSV. On a donc plusieurs moyens de traiter la demande de fichier de l'utilisateur.



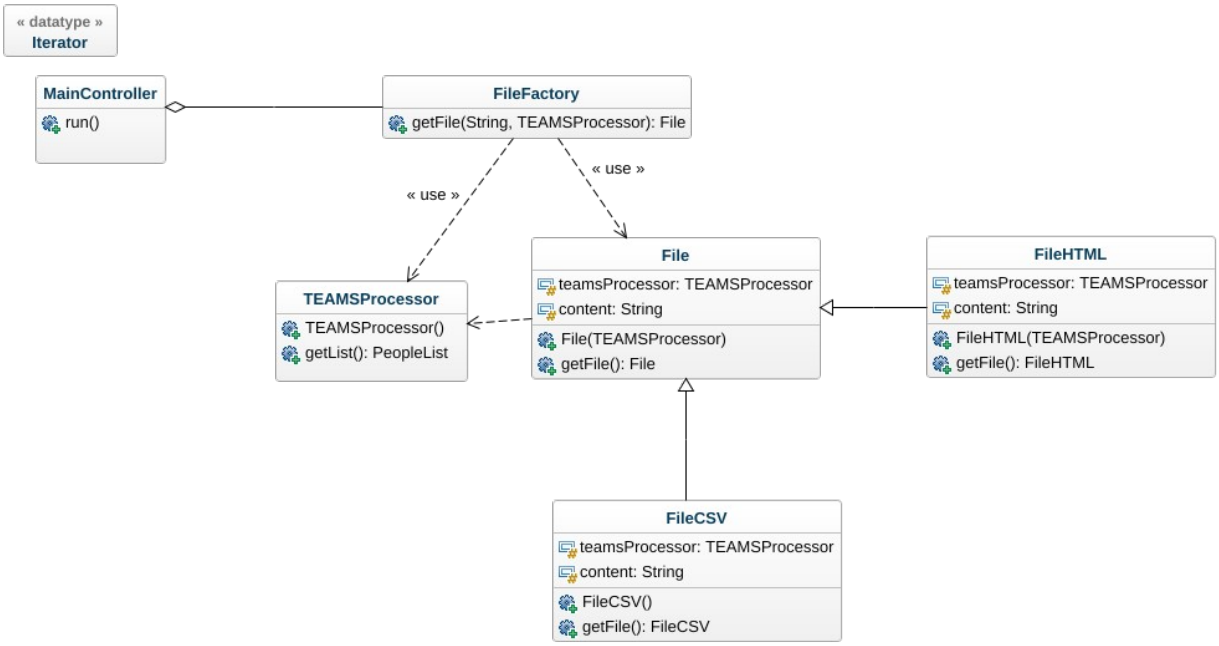


Initialement, le fichier en question est réparti entre deux classes, soit la classe *People*, qui prendra les informations des personnes et les transformera en fichier HTML, et la classe *TEAMSProcessor*, qui renverra, pour chaque People, leurs informations regroupées dans un tableau HTML.

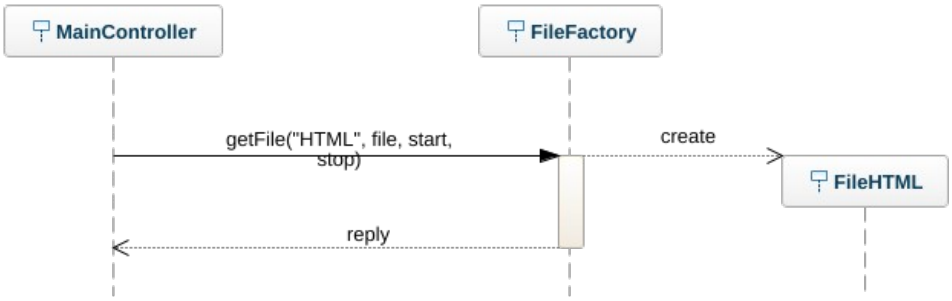
Or, il n'est pas normal que le fichier à générer soit réparti en deux classes qui n'ont rien a voir avec sa création. L'idéal serait donc que le fichier soit créé à partir d'une personne, et non pas qu'une personne créé un fichier à partir d'elle même.

Nous avons donc opté pour une **Factory**, permettant de générer un fichier voulu, sans forcément connaître la classe en question (en effet, l'utilisateur ne connaît que le fichier qu'il souhaite, il n'a pas besoin d'obtenir plus de détails.

En outre, avec une **Factory**, il sera facile d'ajouter plusieurs traitements types, sans pour autant avoir à retoucher au code initial.



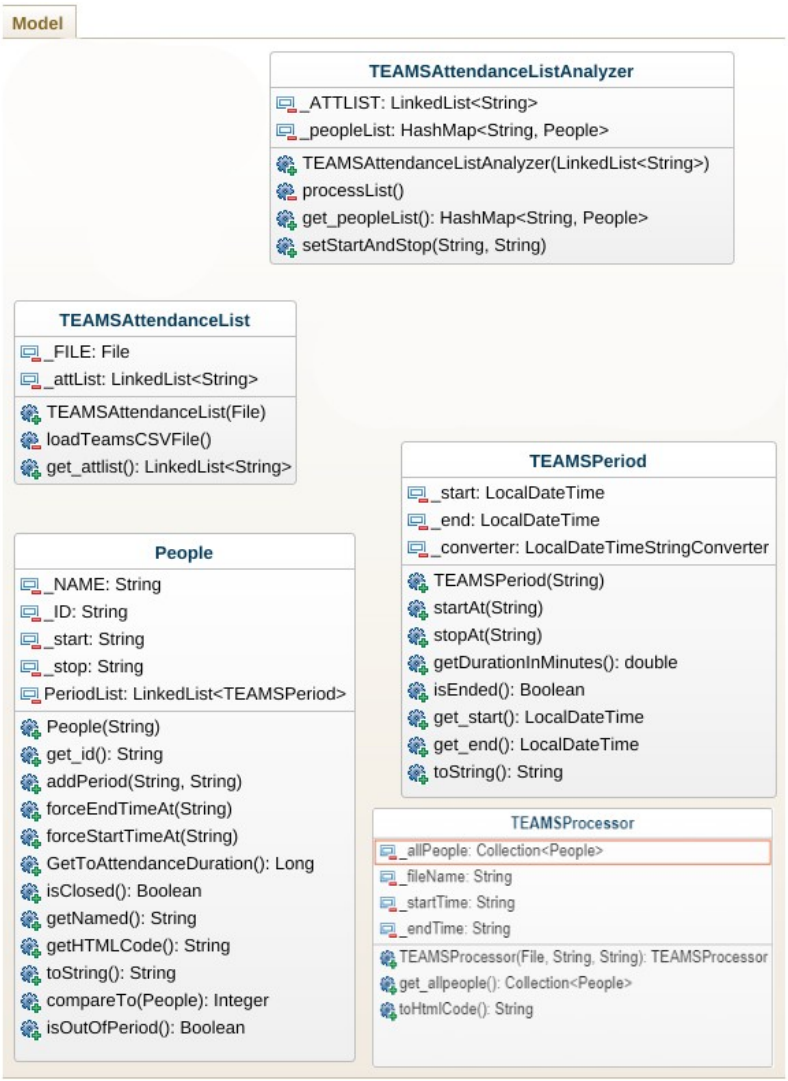
Ici, nous avons une relation entre les package *Model* et *Controller* . Dans le package *Model*, nous avons les classes *FichierHtml* et *FichierCSV* dont le but est de générer un fichier du même nom. Conformément au pattern, ces classes vont hériter d'une classe mère appelée *Fichier* , qui permettra un échange facile lors de la création d'un fichier. Le *Controller* générera alors le bon fichier, en fonction de ce qu'a sélectionné l'utilisateur.



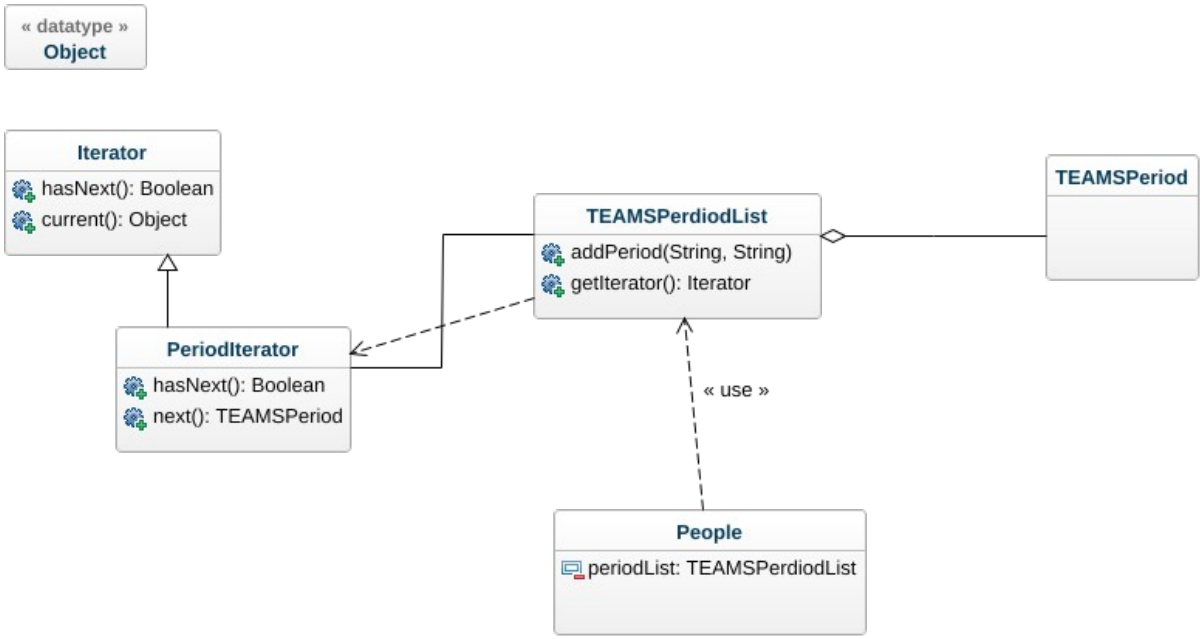
Dans le scénario représenté par le diagramme de séquence ci-dessus, une demande est envoyée, via le *Controller*, à la **Factory**, avec en paramètre le type de fichier à générer à partir du fichier déposé (ici, HTML). Une demande de création sera envoyée à la classe correspondante, pour enfin être donné, au format voulu, au *Controller*.

2. Iterator :

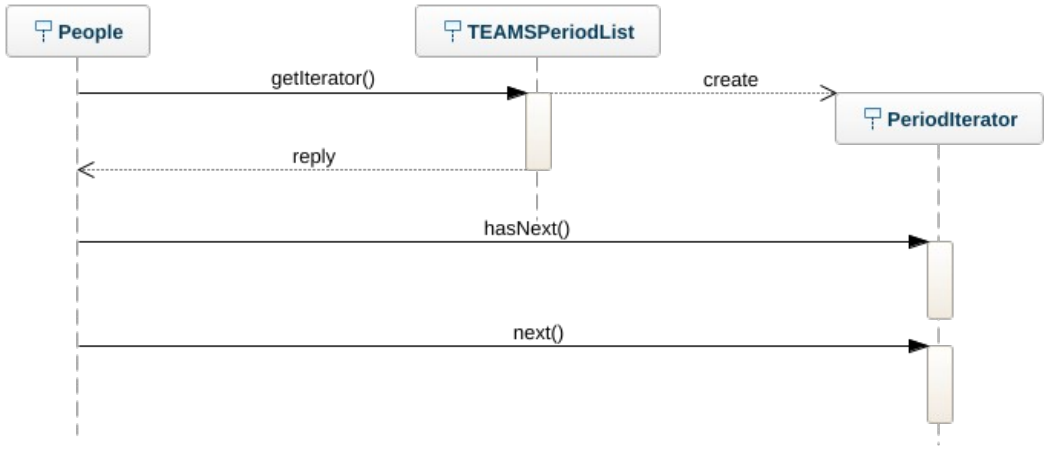
Pour ce patron de conception, nous avons simplement décidé de suivre les consignes.



Initialement, les collections sont parcourues par des **itérateurs** ou non. Dans le 1er cas, ils sont issus directement de *java.util,* et le mécanisme est confondu dans la classe data. Ils sont donc peu permissifs et dépendent de la **data**. Ainsi, nous avons créé nos propres **itérateurs,** dont un exemple d'implémentation est donné par le Diagramme de classe ci-dessous :



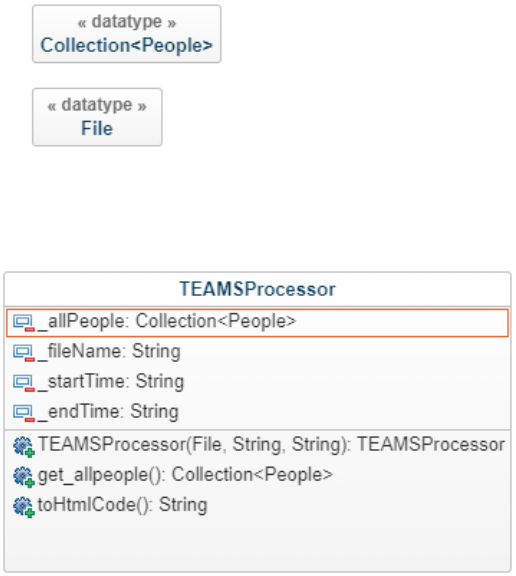
Ici, nous avons l'implémentation de la TEAMS*PeriodList* de la classe *People* , qui sera parcouru par un **itérateur** approprié. Nous utilisons la librairie « java.util.iterator » pour implémenter plus facilement le mécanisme.



Dans le scénario ci-dessus, la classe *People* conçoit son attribut «*_periodList* grâce à la méthode *getIterator()* (implémenté *Iterator* car Override). La classe « PeriodIterator » renverra alors l'itérateur ainsi construit à la classe *People*, en passant par *TEAMSPeriodList*. Ainsi, pour traiter la liste des planning, les méthode *hasNext()* et *next()* seront utilisables.

3. Strategy :

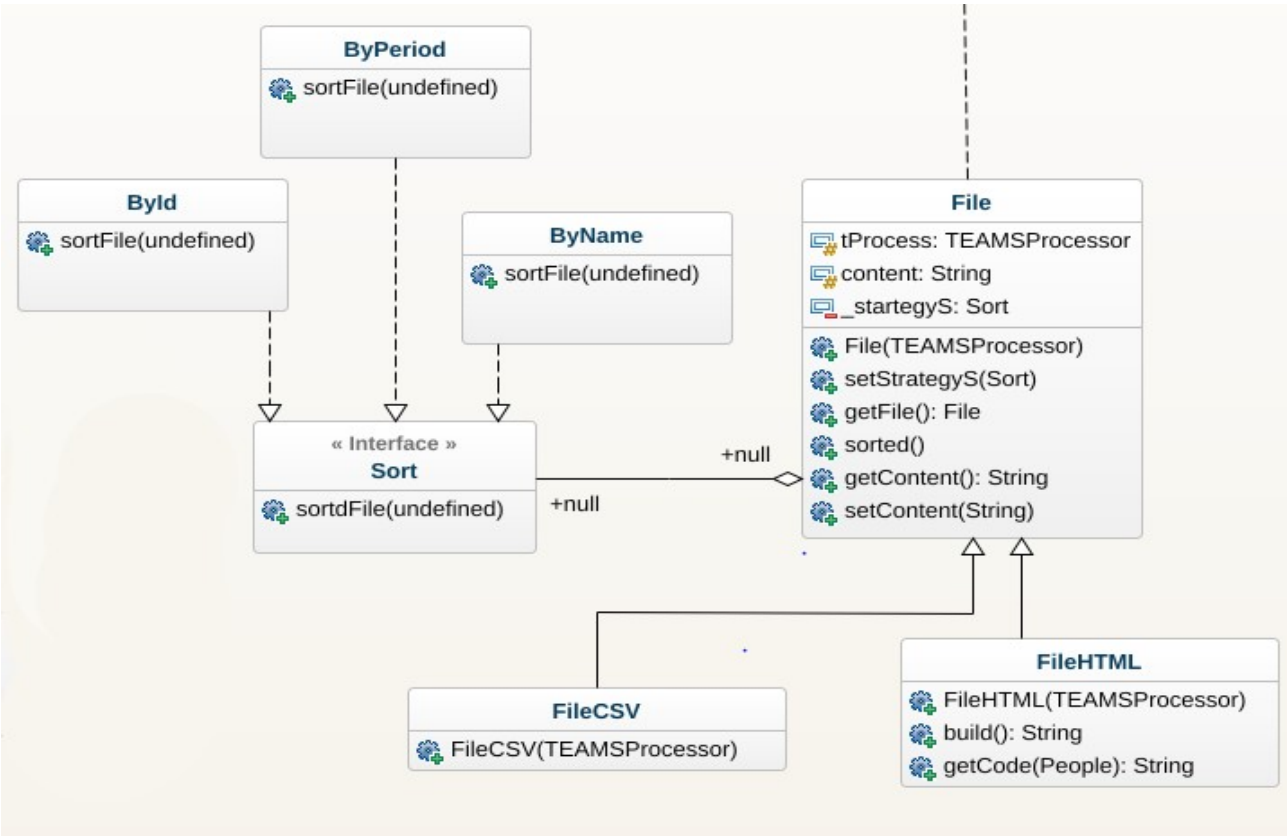
Ici, on nous demande plusieurs moyens pour trier les données dans le fichier.



Initialement, aucune méthode n'existe pour réaliser un tri particulier. *People* implémente l'interface Comparable, mais le tri est réalisé, par défaut, avec l'identifiant d'une personne et dans le constructeur de *TEAMSProcessor*. Il faut donc pour cela ajouter différentes méthodes permettant d'effectuer des tris de manières alternatives.

Or, si nous nous contentons de créer une nouvelle méthodes de tri pour chaque attribut de People, la classe *TEAMSProcessor* risque de se retrouver avec une grande quantité de méthode, apportant ainsi confusion et redondance du code.

Strategy, pattern permettant le choix au niveau des méthodes d'exécution, nous a semblé adapté pour résoudre ce problème :



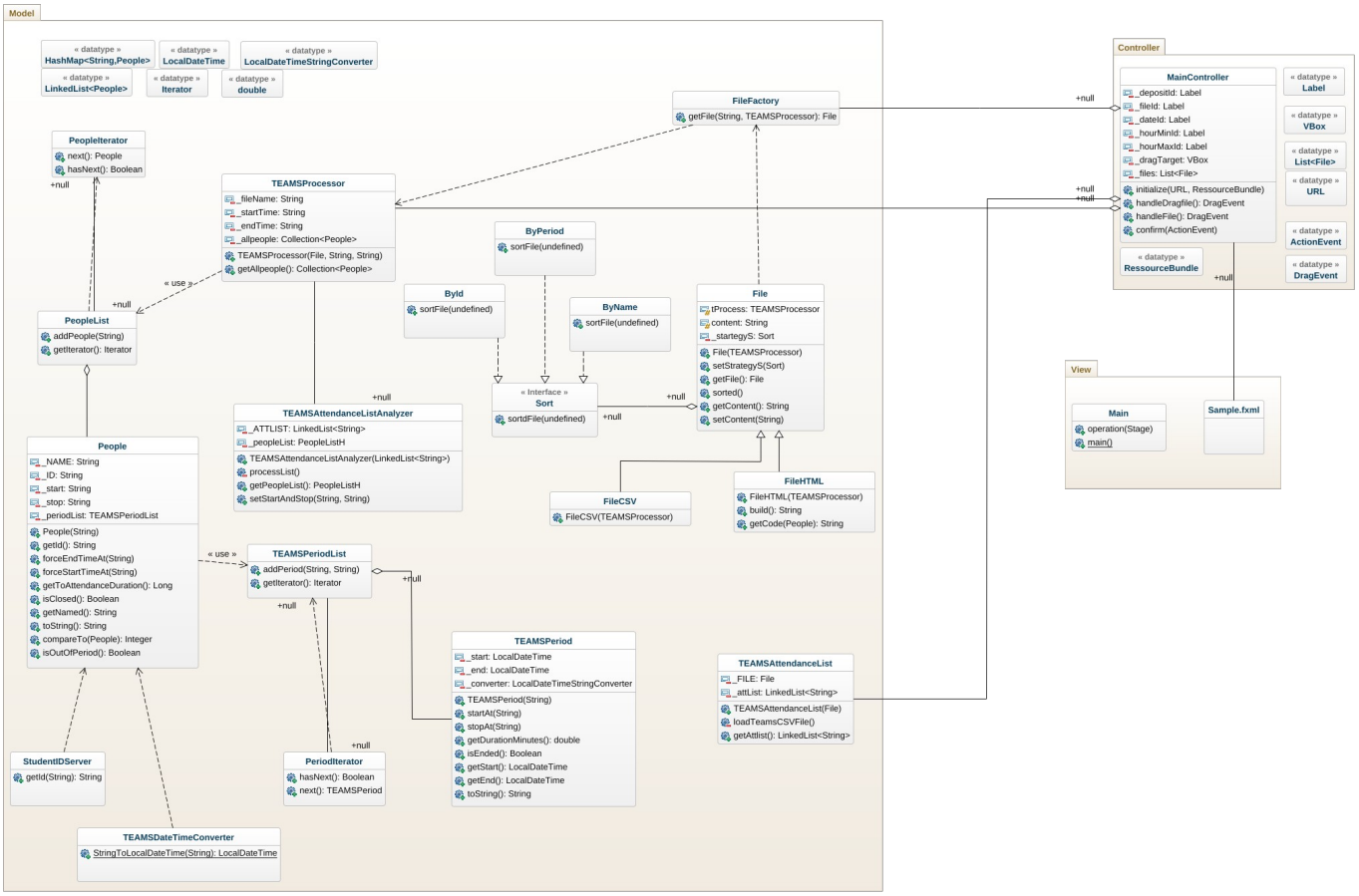
Nous avons ici séparé les différentes possibilités de tris en 3 classes (*ByName, ByPeriod, ById*). Chacune implémentent l'interface *Sort*

Nous avons une relation entre les package M*odel* et *Controller*. Dans le package M*odel*, nous avons les classeurs HTML et CSV dont le but est de générer un fichier du même nom. Conformément au pattern, Ces classes vont implémenter l'interface Sort, qui se chargera de transmettre au Context (*Fichier*), la bonne méthode à utiliser. Le *Controller* générera alors un tri adéquat, en fonction de ce qu'a sélectionné le client.

5 . MVC :

Nous avons décidé de traiter ce point en utilisant un **Model Vue Controller** basique en séparant les classes en 3 packages : Les classes *Model* (*People, TEAMSPeriod, TEAMSAttendanceList, etc...*), les classes *View* (*Main,Sample.fxml*), et les classes *Controller*.

Ici, le rôle du MVC sera de répartir les taches convenablement, afin de ne pas encombrer une classe de méthodes qui ne lui sont pas utiles.



Dans ce cas de figure, nous avons 3 types de comportements différents, que l'on va qualifier **Observer** (*Vue*) , **Observable** (*Model*) et **Listener** (*Controller*). Oh un **pattern (Observer)** ! Respectivement, le rôle de chaque entité sera :

-Observer : Examinera le modèle pour renvoyer le comportement de ce dernier à l'utilisateur

-Observable : Contiendra l'ensemble des données à afficher.

-Listener : Fera office d'intermédiaire, permettant un traitement du modèle grâce au fichier reçu par la vue.

6. Pas de Pattern

Dans le cadre d'un ajout de fonctionnalité de recherche par nom et prénom de l'user, il suffira, le moment voulu, d'ajouter des *textFields* à la vue, pour lesquels on récupérera le texte via un getter, pour parser ce dernier via la fonction *equalsIgnoreCase()* . Ainsi, les noms et prénoms récupérés seront traités par le *controller*, et ce quelque soit leur taille (majuscule ou minuscule).

II / Les points débattus:

Devons nous séparer le Model de la Data pour le MVC ?

Étant donné le coté métier qu'offrent certains pattern ,nous avons soulevé l'idée de pouvoir séparer les classes métiers et data. Or, pour un projet de cette envergure, ce choix nous à paru un peu trop anecdotique pour obliger son application.

Un décorateur pour le point n°4?

En voyant les contraintes du 4, nous avons de prime à bord pensé à utiliser un **Décorateur.** Ainsi, à la création , il aurait été possible d'encapsuler plusieurs objets, eux mêmes déterminés par le *Controller* (Par exemple, créer new FichierNom(New fichier ID()) etc.).

Cependant, nous avons été vite limité par la conception de base du projet. Si nous voulons créé un **Décorator**, il est nécessaire de passer par un interface, qui sera appelé lors de l'instanciation de l'objet. Or, dans notre cas, ce pattern ce serait appliqué sur la classe fichier HTML, servant de classe concrète pour notre Fabrique. En outre, il nous est également venu à l'esprit d'utiliser un Strategy, mais nous nous retrouvons dans une impasse similaire au final.

Nous avons donc opté pour une neutralisation du CSS, en concaténant le contenu du fichier HTML ainsi créé.

Liens :

https://app.genmymodel.com/editor/edit/_bgFJcKaYEeuXgcfNRvw02A?locale=fr#

*Illustrations sur la session GenMyModel de **Gauthier ROUSSEL***