Rapport LOG1410

1)

Pour le premier patron nous avons voulu contrôler la classe Performance. En effet, nous avons estimé qu'il fallait trouver un critère d'accès à la classe afin de contrôler l'accès à celui-ci. Nous avons choisi ce patron afin que l'élève puisse écouter sa performance lorsqu'il le souhaite car la performance est une classe lourde qui dépend de plusieurs agrégats tel que la bande sonore, le rythme, les nuances…Ce patron est en lien avec notre cas d'utilisation « Évaluer la performance ». En effet, dans ce cas d'utilisation l'élève enregistre sa performance.

Pour le second patron, nous avons décidé d'utiliser le patron itérateur afin de pouvoir itérer sur les structures de fichiers des élèves qui comportent l'évolution de ces derniers. Le professeur aura accès à l'interface des fichiers et par conséquent à l'évolution de chaque élève. Ce patron est encore une fois en lien avec le cas d'utilisation Évaluer la performance car après l'analyse de cette dernière vers le cloud, une rétroaction de la progression de l'élève sera donnée au professeur.

En ce qui concerne le troisième patron, nous avons décidé d'utiliser le patron singleton afin de pouvoir avoir une instance unique de la partition numérique. En effet, lorsque le professeur enverra une même partition numérique à tous les élèves, ceux-ci devraient tous avoir la même partition numérique donc une seule instance est suffisante

On a finalement décidé d'utiliser le patron observateur en guise de dernier patron. En effet, on souhaite que lorsque la performance de l'élève est terminée, celle-ci notifie la mémoire et le cloud de stocker ou synchroniser la partition selon si la connexion est stable ou non. Ce patron est utile dans cette situation car on veut que les classes mémoire et cloud suivent la classe performance de l'élève.

Singleton avec élève et partition numérique (C.U : Optimiser l'apprentissage)

Observateur : Enregistrement(class performance) finit —-> notifier l'analyse (class cloud) grâce à des méthodes.
(C.U : Evaluer la performance)

Iterateur :
historique d'évolution,

2)

Patron Proxy:

Avantages:

Le premier avantage du patron Proxy est le principe ouvert/fermé. En effet, il est possible d'ajouter des proxy à d'autres méthodes de la classe Performance sans que cela impacte le client. On pourrait faire en sorte que le ProxyPerformance puisse contrôler l'envoi d'une performance vers le Cloud afin que celle-ci soit analysé.
Le deuxième avantage est de contrôler l'objet du service sans que le client ne s'en aperçoive. Concernant l'exemple mentionné précédemment, le ProxyPerformance pourra contrôler l'envoi de la performance en fonction de la perte de connexion.
Le proxy peut être un patron lourd si nous voulons contrôler d'autres objets autre que l'enregistrement et la lecture de ce dernier.

Patron Iterateur :

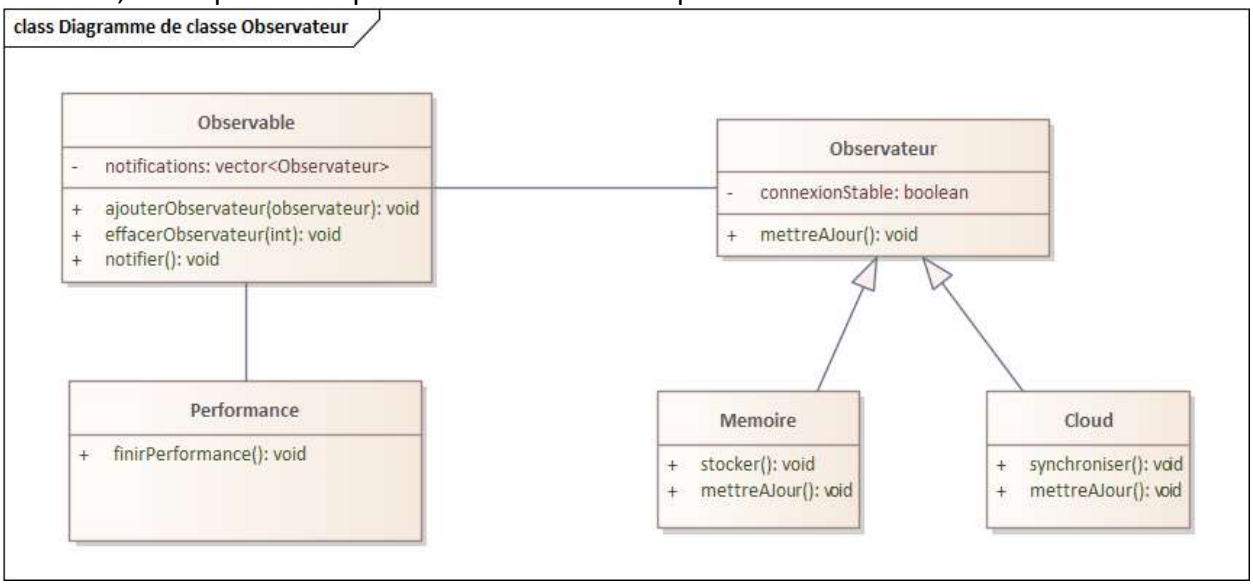
Le patron itérateur a un premier avantage qui est la réutilisation de la classe ItérateurConcret. En effet, avec la classe Itérateur on peut parcourir plusieurs types de conteneurs. Dans notre cas, nous voulions seulement parcourir les structures de fichiers. Si nous voulons parcourir un autre type de collections comme un tableau d'ajustement des objectifs de chaque élève, il faudra alors rendre la classe IterateurConcret une classe template.
Le second avantage est que nous pouvons manipuler l'itérateur comme on le souhaite. En effet, le professeur peut parcourir les structures de fichiers un par un mais aussi revenir en arrière lorsqu'il en a besoin.
Un troisième avantage est la simplification de l'interface des structures de fichiers. On pourrait penser à parcourir ces derniers en appuyant sur un bouton au lieu de les avoir tous afficher sur l'interface.
L'inconvénient principal de patron est d'implémenter une classe IterateurConcret pour des conteneurs qui sont simples. Ici la structure de fichier comporte seulement les informations de l'élève ainsi que leur évolution mais en réalité celle-ci est plus complexe. L'utilisation de l'itérateur dans notre cas est pertinente. Cependant, dans l'implémentation, nous avons mis des éléments simples dans les fichiers d'évolution donc on peut penser que l'itérateur n'est pas efficace. Or, en réalité ces structures sont beaucoup plus complexes.

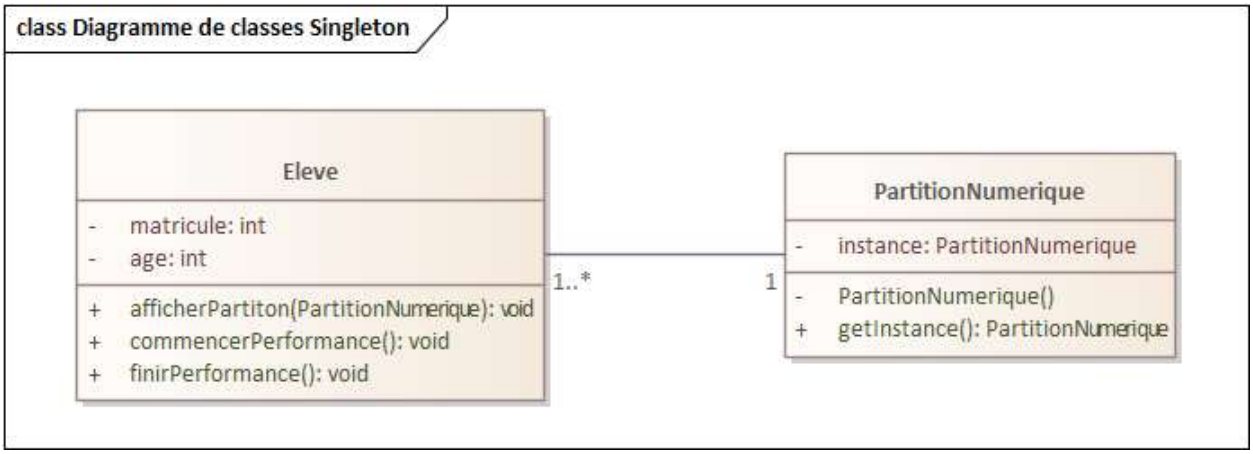
Patron Singleton:
Le premier avantage du patron singleton est de permettre l'unicité de l'instance de notre classe, dans notre cas la classe PartitonNumerique. En effet, il est impossible de créer un objet PartitionNumerique car son constructeur est "delete" et donc inaccessible. Il faudra donc appeler un getter qui permet d'accéder à l'instance en attribut privé. De plus, le second avantage est que l'objet ne peut être initialisé que la première fois ou il est appelé, ce qui permet d'empêcher l'instanciation d'autres objets de type PartitionNumerique.
Cela permet aussi d'obtenir un point d'accès global à cette instance, ce qui facilite le contrôle et la manipulation de cette dernière.

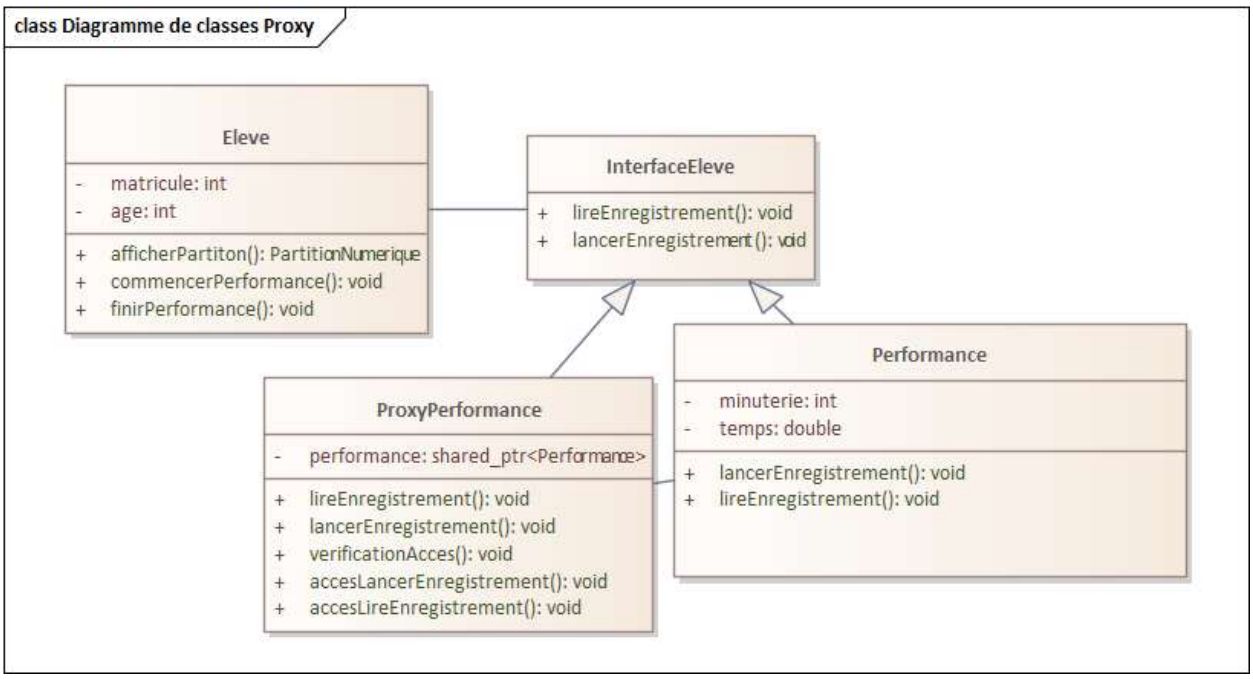
Un inconvénient de taille de ce patron est que celui-ci ne respecte pas le principe de responsabilité unique, qui est fondamental. Aussi, il faut veiller à ce que le singleton ne se retrouve pas en plusieurs exemplaires, ce qui lui perdrait son utilité. Enfin, en terme d'optimisation, il peut être difficile de tester le singleton car le constructeur de ce dernier est privé et rends les tests unitaire très difficiles

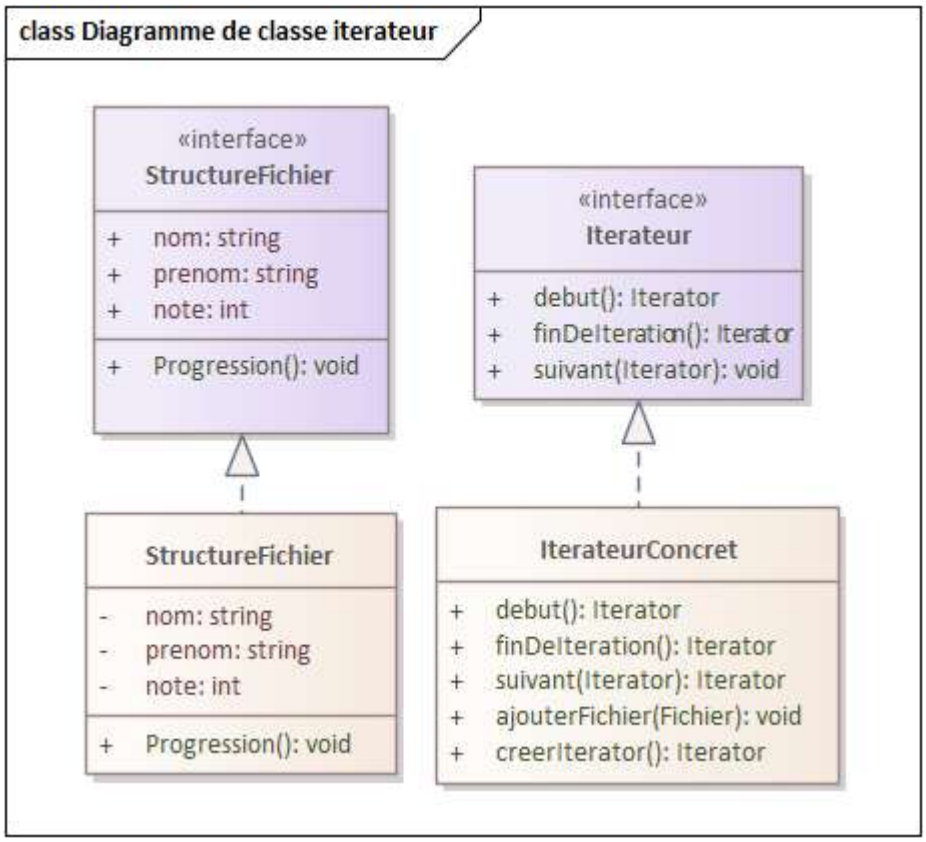
Patron Observateur:
Le patron Observateur est en effet très intéressant si on veut que les modifications de l'état d'un objet en impacte d'autres en les "notifiant". Un avantage de ce patron est qu'on peut ajouter des classes notifiées sans pour autant avoir à toucher à l'objet qui notifie. En effet, on crée dans la classe un attribut privé de type vecteur qui contient toutes les classes qu'on va notifier. On peut ainsi en ajouter a notre guise à l'aide d'une méthode adaptée.

L'inconvénient principal de ce patron est que les classes notifiées seront avertis de manière aléatoire, on ne peut donc pas déterminer un autre précis de notifications.






class Diagramme de classes Proxy
Eleve
- matricule: int
- age: int
+ afficherPartiton(): PartitionNumerique
+ commencerPerformance(): void
+ finirPerformance(): void
InterfaceEleve
+ lireEnregistrement(): void
+ lancerEnregistrement(): void
ProxyPerformance
- performance: shared_ptr<Performance>
+ lireEnregistrement(): void
+ lancerEnregistrement(): void
+ verificationAcces(): void
+ accesLancerEnregistrement(): void
+ accesLireEnregistrement(): void
Performance
- minuterie: int
- temps: double
+ lancerEnregistrement(): void
+ lireEnregistrement(): void


class Diagramme de classe iterateur
«interface»
StructureFichier
+ nom: string
+ prenom: string
+ note: int
+ Progression(): void
«interface»
Iterateur
+ debut(): Iterator
+ finDeIteration(): Iterator
+ suivant(Iterator): void
StructureFichier
- nom: string
- prenom: string
- note: int
+ Progression(): void
IterateurConcret
+ debut(): Iterator
+ finDeIteration(): Iterator
+ suivant(Iterator): Iterator
+ ajouterFichier(Fichier): void
+ creerIterator(): Iterator