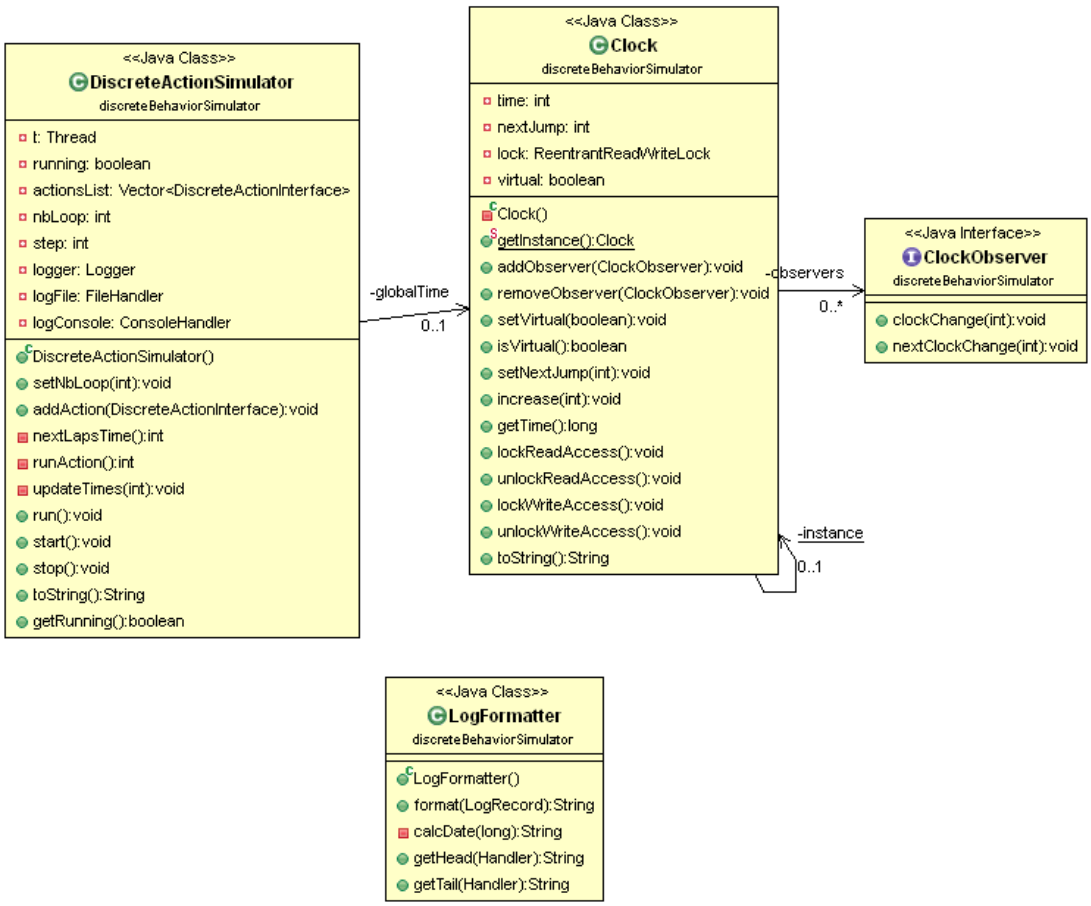
Reverse Engineering

Package: DiscreteBehaviorSimulator

Diagramme UML



Au niveau de ce package, nous avons donc la présence de 3 Classes et d'une interface :

-Classe DiscreteActionSimulator : Cette classe permet de lancer le simulator sous forme d'un Thread.

-Classe Clock : Cette classe utilise le pattern singleton et permet d'instancier une horloge générale, permettant de gérer l'exécution des différents éléments « Action » ou « ClockObserver ».

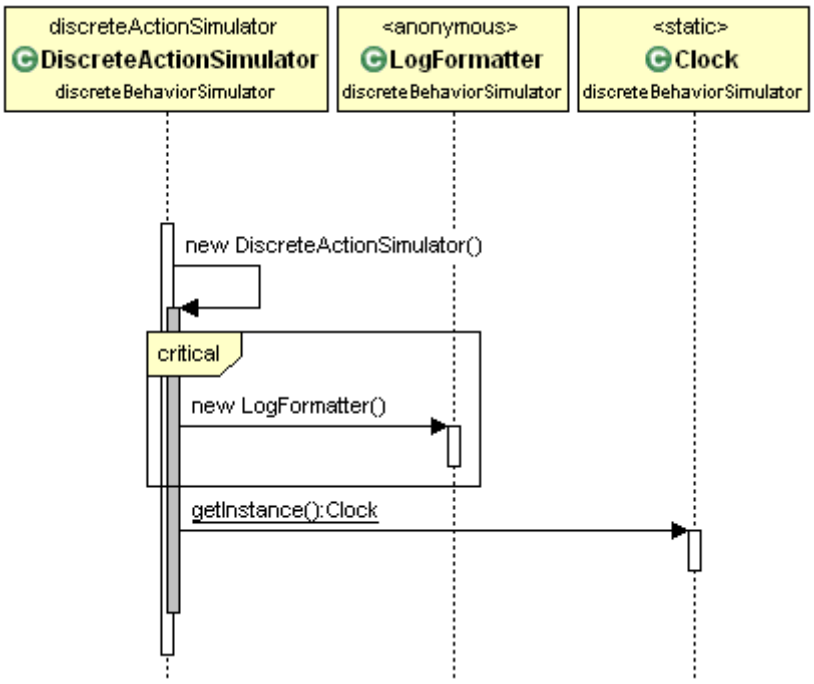
-Classe LogFormatter : Permet la création de différents logs (écriture dans un fichier .log, écriture dans la sortie standard…)

-Interface ClockObserver : Permet l'utilisation de ce package pour d'autres projets. Si ce package est utilisé pour d'autres projets, les classes devront implémenter cette interface.

Diagramme de séquence

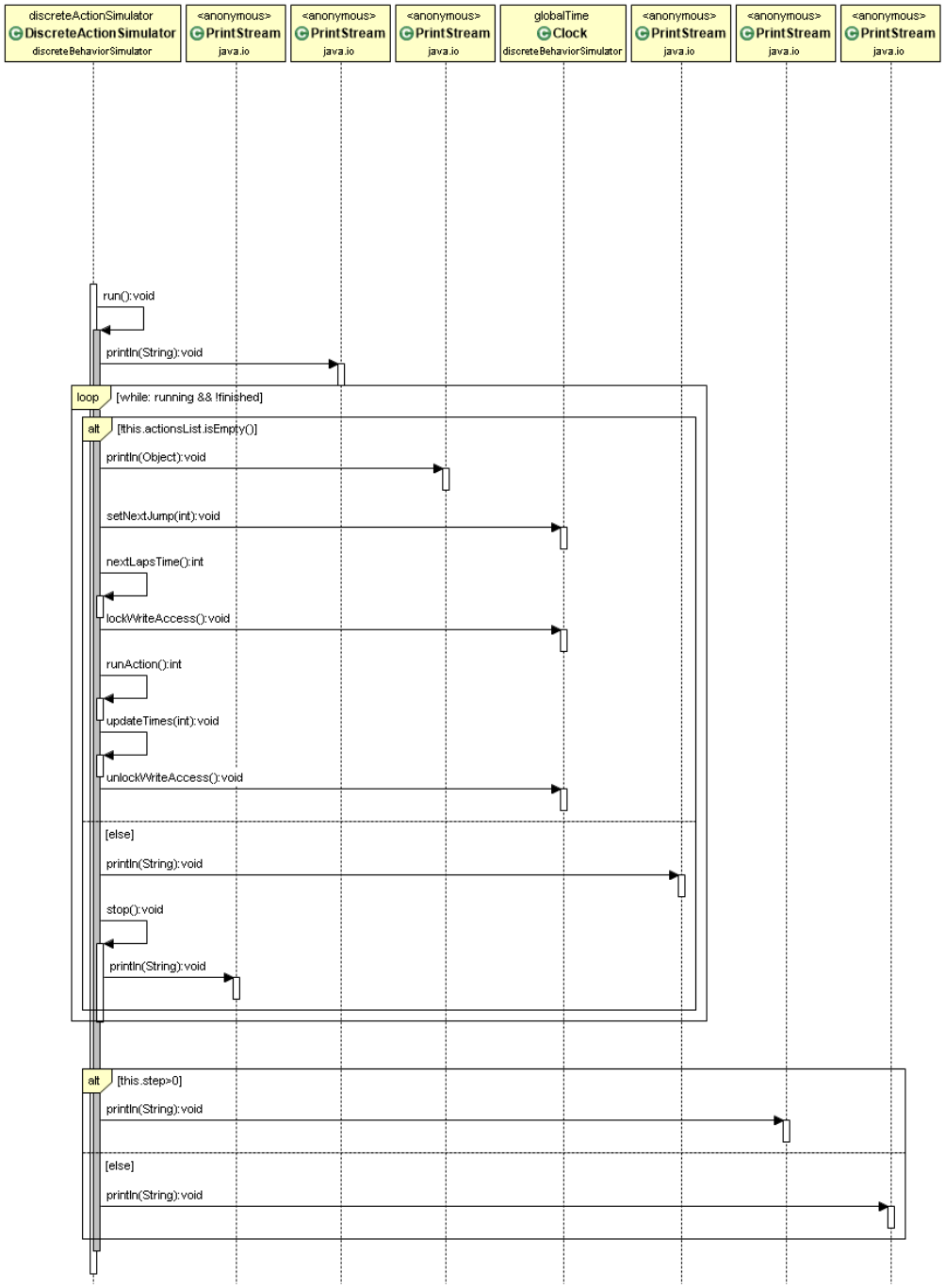
Pour cette partie nous allons principalement étudier les diagrammes de séquence des méthodes appartenant à la classe DiscreteActionSimulator.

Diagramme séquence : Le constructeur



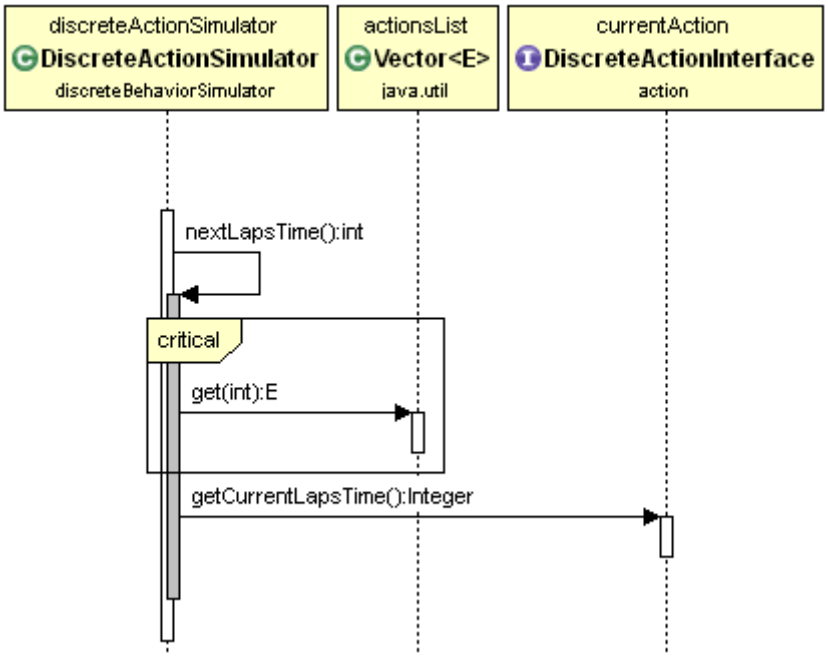
Cette classe de simulateur s'instancie en récupérant une instance de la classe LogFormatter. Puis elle récupère une instance de Clock qu'elle met dans la variable globalTime. Enfin, elle se définit comme nouveau Thread.

Diagramme séquence : Méthode run()



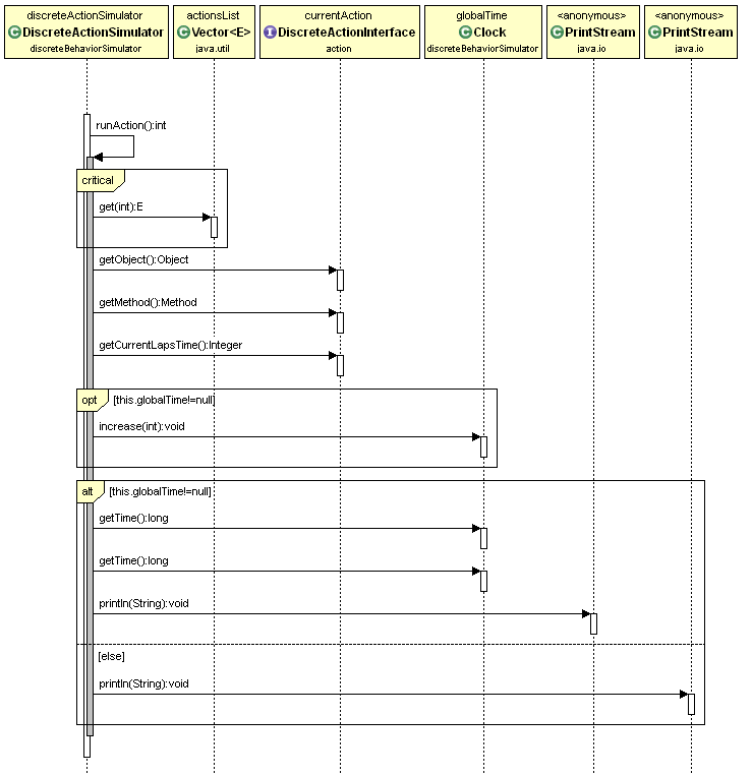
La méthode run est une méthode qui se lance à l'issue de la méthode start(). La méthode run() permet de lancer l'exécution du simulateur et c'est à l'intérieur de celle-ci que qu'il y aura l'interaction avec le package action .Elle présente une boucle while qui s'exécute tant que les actions définies ne sont pas finies (elle possède une liste d'action). Si la liste d'action est vide, le programme s'arrête, sinon il s'exécute. A chaque itération de la boucle, si la liste n'est pas vide, elle va exécuter trois méthodes : nextLapsTime(), runAction() et updateTime(int). Le détail des différentes méthodes et de leurs diagrammes est présenté dans la prochaine section.

Diagramme séquence : Méthode nextLapsTime()



Cette méthode permet de récupérer le laps de temps associé à la l'action présente dans la tête de la liste d'action (l'action courante) et va l'injecter dans le global time de DiscretActionSimulator. Cela permet par la suite de verrouiller l'accès à la mémoire pendant la durée de l'action en cours avec la méthode lockWirteAcces() du globalTime.

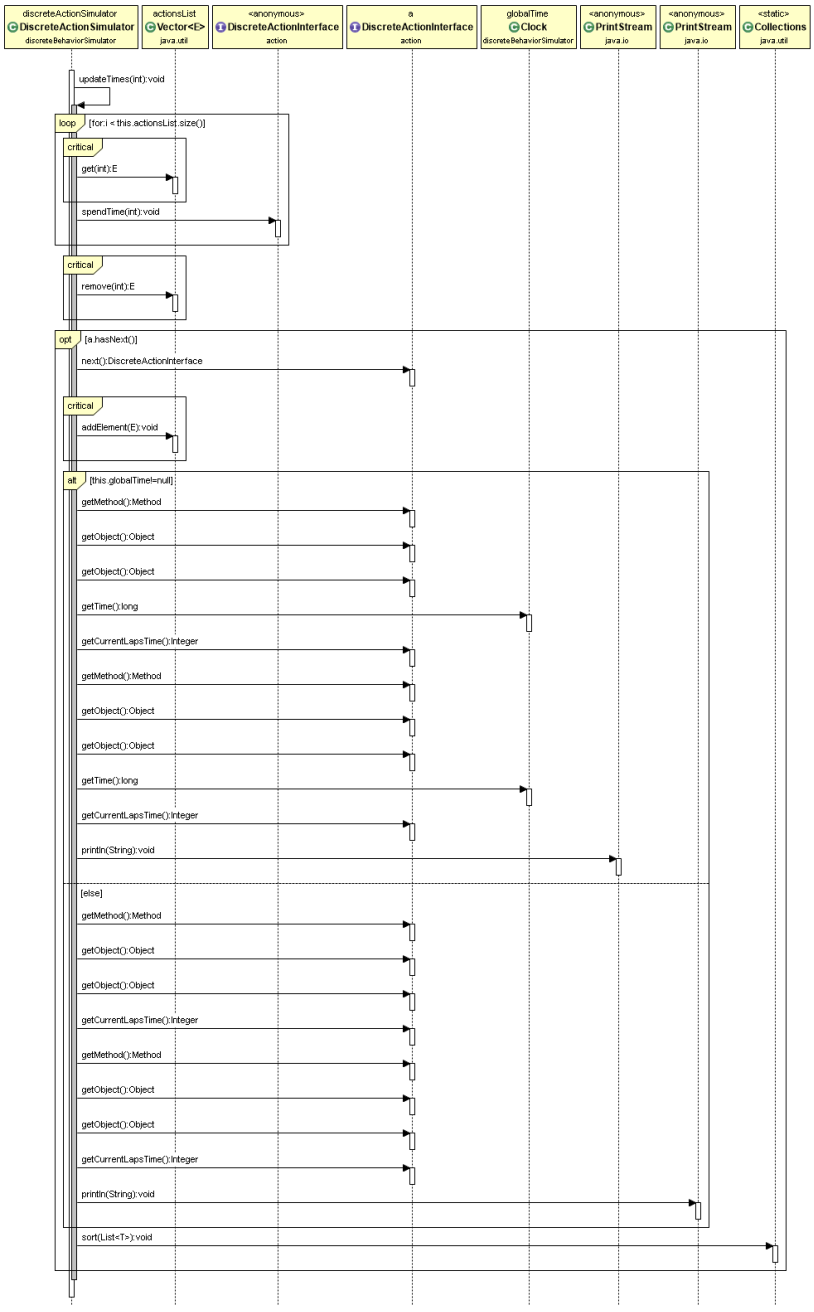
Diagramme séquence : Méthode runAction()



Cette méthode permet de récupérer l'action à exécuter qui est composée d'une classe, d'une méthode de cette classe ainsi que le temps que va mettre l'action à s'exécuter (LapsTime). Elle va mettre à jour l'horloge en incrémentant le temps courant avec le LapsTime.

Elle affiche sur l'entrée standard qu'elle action est en train d'être exécuté, à quel moment elle est exécutée et après si c'est après un laps de temps ou non.

Diagramme séquence : Méthode updateTimes()



Cette méthode a deux fonctionnalités. La première permet de pouvoir actualiser l'ensemble des laps de temps de chaque action en fonction du temps d'exécution de l'action courante. Cela permet de savoir au bout de combien de temps l'action va être effectuée. Si par exemple, on prend deux actions avec un timer périodique de fréquence T (même timer pour les deux actions), alors la première action va s'effectuer au bout d'un temps T et la deuxième action va s'exécuter au bout de T-T, c'est à dire 0.

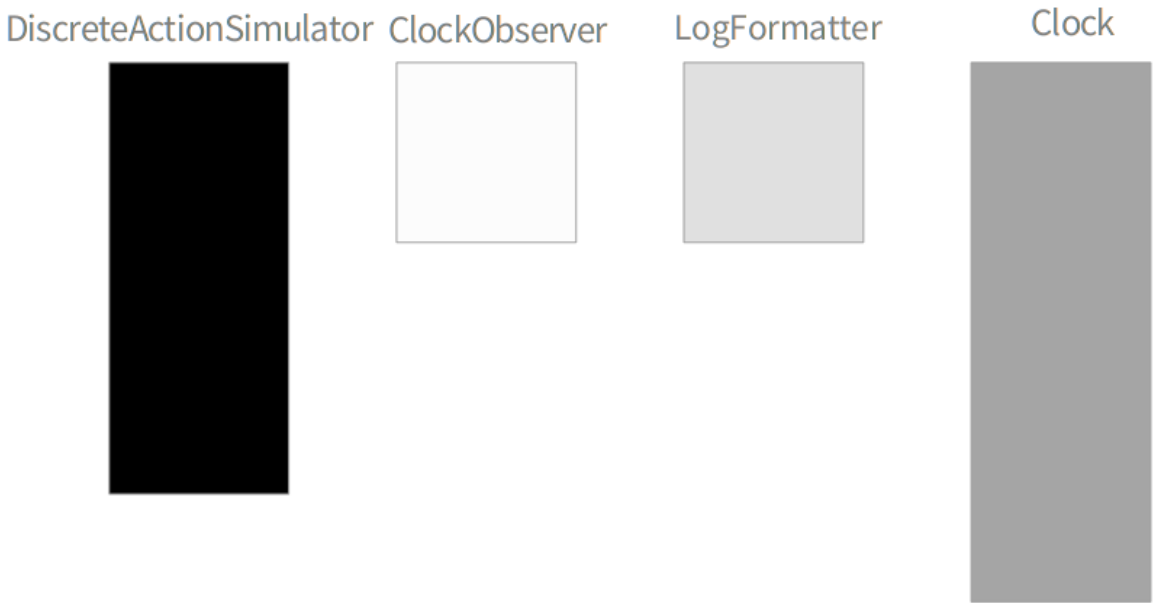
La deuxième fonctionnalité permet dans le cas d'une action possédant un timer qui n'est pas encore finis, d'être enlevée de la tête de la queue et d'être remplacée à la fin de celle-ci. Par exemple, une action avec un périodique timer suivra le schéma précédant. Pour le cas d'une action qui possède un « One Shot timer », l'action ne sera pas remise dans la queue.

Résumé de la séquence d'exécution

Le simulateur possède une liste d'action « ActionList » (implémentant « DiscreteActionInterface »). Chaque action de cette liste peut posséder une liste de « timmer » à exécuter. Tant que la liste d'action n'est pas vide, l'exécution s'effectue comme cela :

- Récupération du laps de temps correspondant à la tête de la liste
- Ferme l'accès a la mémoire pour les autres actions
- Lance l'action qui va permettre d'exécuter la méthode de l'objet ce trouvant dans l'action
- Mettre à jour les temps et si l'action courante possède plusieurs temps d'exécution (« timmer »), elle va remettre l'action courante dans la liste d'action, sinon elle supprime la tête (courante, pour laisser la place à la prochaine action).

Reverse Enginneering avec Moose



En utilisant Moose, nous pouvons avoir un aperçu de l'état actuel du package discreteBehaviorSimulator de manière quantitative où chaque rectangle représente une classe du package. En effet, nous pouvons visualiser le nombre d'attributs (largeur du rectangle), le nombre de méthodes (longueur du rectangle) et enfin le nombre de lignes de code (blanc : peu de lignes, noir : beaucoup de lignes). Nous pouvons déduire que la classe DiscrèteActionSimulator possède beaucoup de lignes relativement au nombre de méthodes. De ce fait, cela pourrait nous permettre d'axer le refactoring en priorité sur cette classe car elle peut contenir plusieurs problèmes :

- Présence de ligne de code laissées en commentaire (on peut penser un oubli de la part du développeur)
- Beaucoup de lignes de code par méthodes, ce qui peut poser des problèmes pour la modularité du code (également impacter la notion d'atomicité).