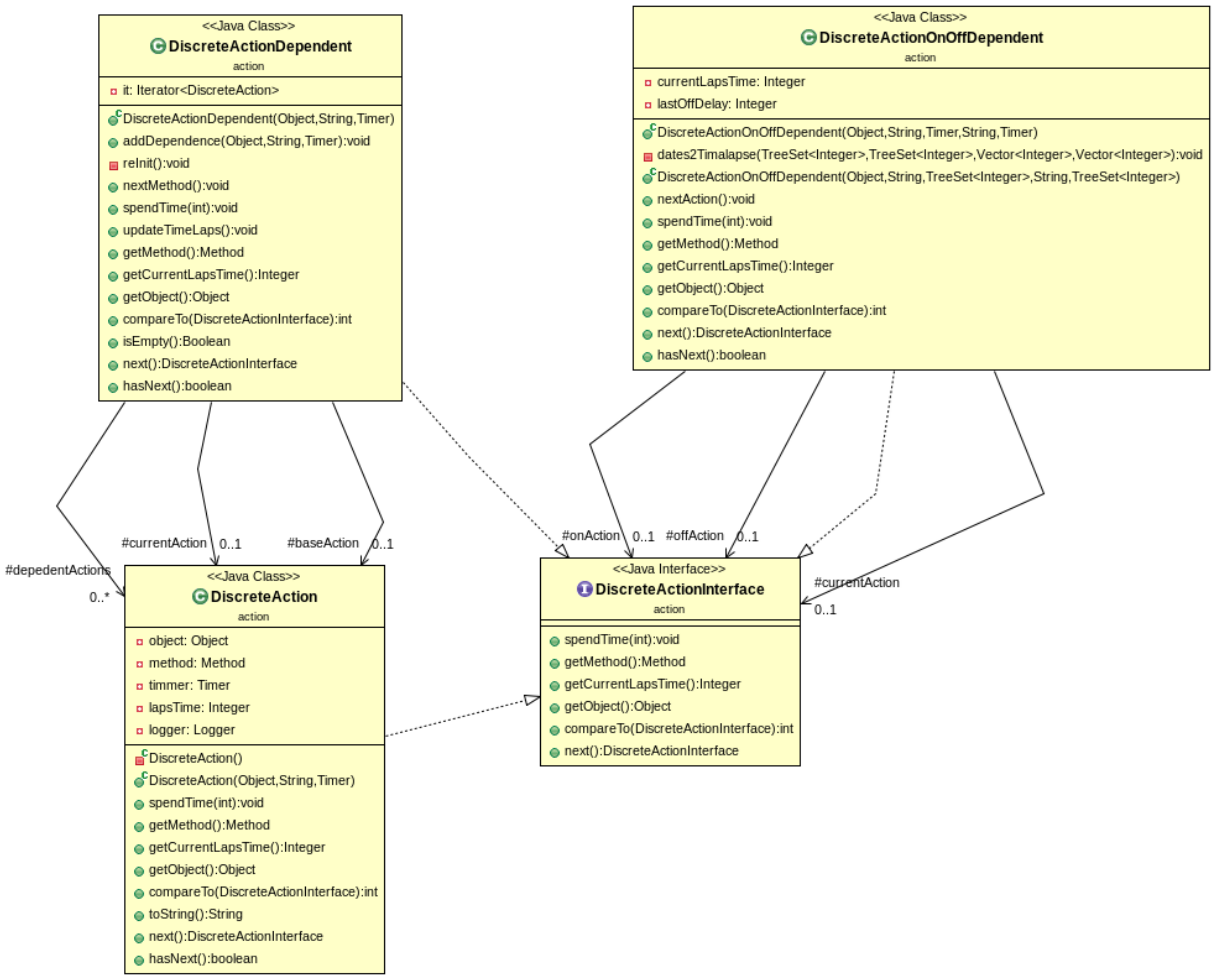
Reverse Engineering

Package: action

Diagramme UML



Dans ce package, il y a 3 classes et une interface :

- Classe DiscreteAction : Permet de créer et gérer une action discrète.

- Classe DiscreteActionDependant : Permet de créer et gérer une série d'actions dépendantes les unes des autres.

- Classe DiscreteActionOnOffDependant : Permet de créer et gérer une série d'action dépendantes les unes des autres, séparé en action On et action Off, afin d'alterner entre les actions On et les actions Off.

- Interface DiscreteActionInterface : Permet l'utilisation de ce package pour d'autres projet. Bien sûr toutes les autres classes doivent implémenter cette interface.

Diagramme de séquence

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

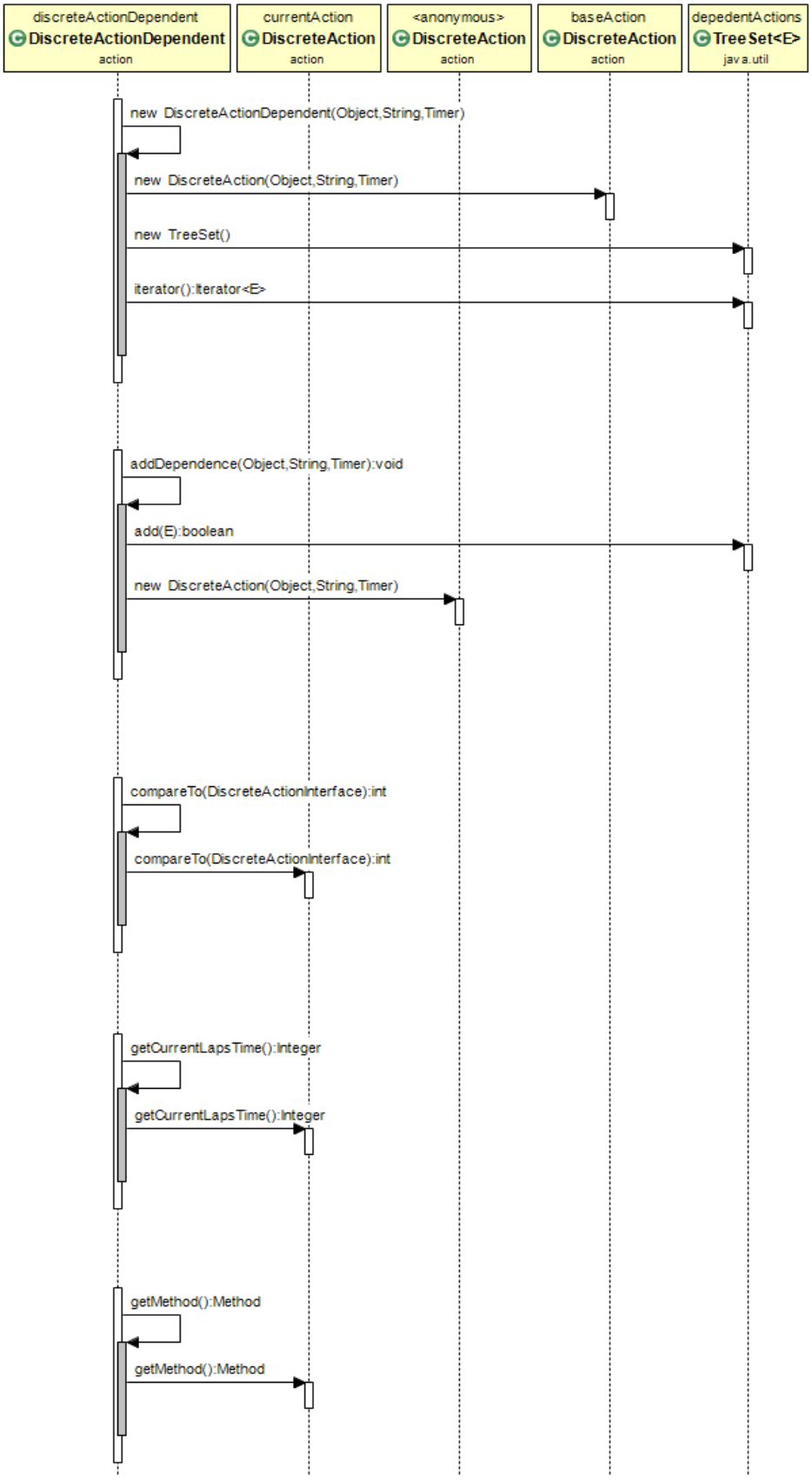


Diagramme séquence : Le constructeur

Cette classe s'instancie avec pour paramètres un objet o, une string baseMethodName et un timer timberBase, qui permettent de créer l'action de base. Une nouvelle instance de classe DiscreteAction est donc créée à partir des paramètres, c'est l'action de base, la première qui est appelée, associée à la variable baseAction (this.baseAction). Ensuite une nouvelle instance de TreeSet est créée, qui contient des actions, associée à la variable depedentAction (this.depedentAction). Ensuite, l'iterator du TreeSet depedentAction est récupéré et associé à la variable it (this.it).

Diagramme séquence : Méthode addDependence

La méthode addDependence prend pour paramètre un objet o, une string depentMethodName et un timer timerDependence. Elle permet de créer une nouvelle action avec ces paramètres et l'ajouter au TreeSet depedentAction.

Diagramme séquence : Méthode compareTo

Cette méthode permet de comparer le lapsTime du currentAction (this.currentAction, initialement égale à this.baseAction) au dernier last time sans update (méthode getCurrentLapsTime) par rapport à un paramètre c de classe DiscreteActionInterface. La méthode compareTo appel la méthode compareTo du currentAction, avec pour paramètre l'interface c. Un entier est retourné en fonction du résultat de la comparaison entre le lapsTime du currentAction et de la valeur retournée par la méthode getCurrentLapsTime.

Diagramme séquence : Méthode getCurrentLapsTime

Cette méthode retourne la valeur (entier) retournée par la méthode getCurrentLapsTime de currentAction.

Diagramme séquence : Méthode getMethod

Cette méthode retourne la valeur (Method) retournée par la méthode getMethod de currentAction.

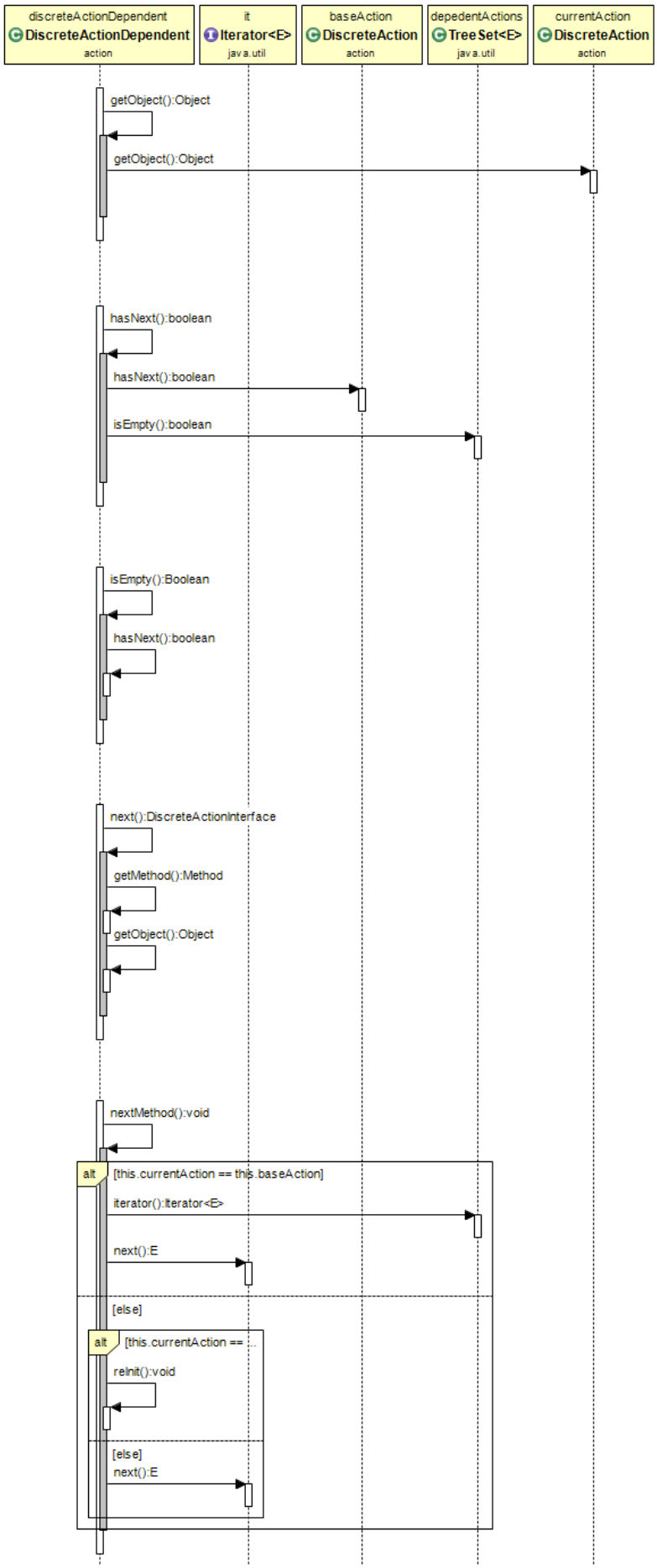

discreteActionDependent
DiscreteActionDependent
action
it
Iterator<E>
java.util
baseAction
DiscreteAction
action
depedentActions
TreeSet<E>
java.util
currentAction
DiscreteAction
action
getObject():Object
getObject():Object
hasNext():boolean
hasNext():boolean
isEmpty():boolean
isEmpty():Boolean
hasNext():boolean
next():DiscreteActionInterface
getMethod():Method
getObject():Object
nextMethod():void
alt
[this.currentAction == this.baseAction]
iterator():Iterator<E>
next():E
[else]
alt
[this.currentAction ==
reInit():void
[else]
next():E

Diagramme séquence : Méthode getObject

Cette méthode retourne la valeur (Object) retournée par la méthode getObject de currentAction.

Diagramme séquence : Méthode hasNext

Cette méthode retourne un booléen, permettant de savoir s'il y a encore une action dans depedentAction ou dans le timer de baseAction. Elle récupère le booléen retourné par la méthode hasNext de baseAction et le booléen retourné par la méthode isEmpty de depedentAction (TreeSet). Le booléen retourné est True si baseAction.hasNext() retourne True ou que depedentAction.isEmpty() retourne False.

Diagramme séquence : Méthode isEmpty

Cette méthode retourne un booléen qui est l'inverse du résultat de hasNext.

Diagramme séquence : Méthode next

Cette méthode retourne une discreteActionInterface après avoir fait appel à ses méthodes getObject et getMethod.

Diagramme séquence : Méthode nextMethod

Cette méthode ne retourne rien (void). Si currentAction est baseAction, alors la méthode récupère l'iterator de dependentAction et associe la prochaine action (it.next()) à currentAction. Si currentAction est l'ancien élément de depedentAction (depedentAction.last()), alors la méthode reInit est appelée. Sinon, currentAction est associée à la prochaine action (it.next()).

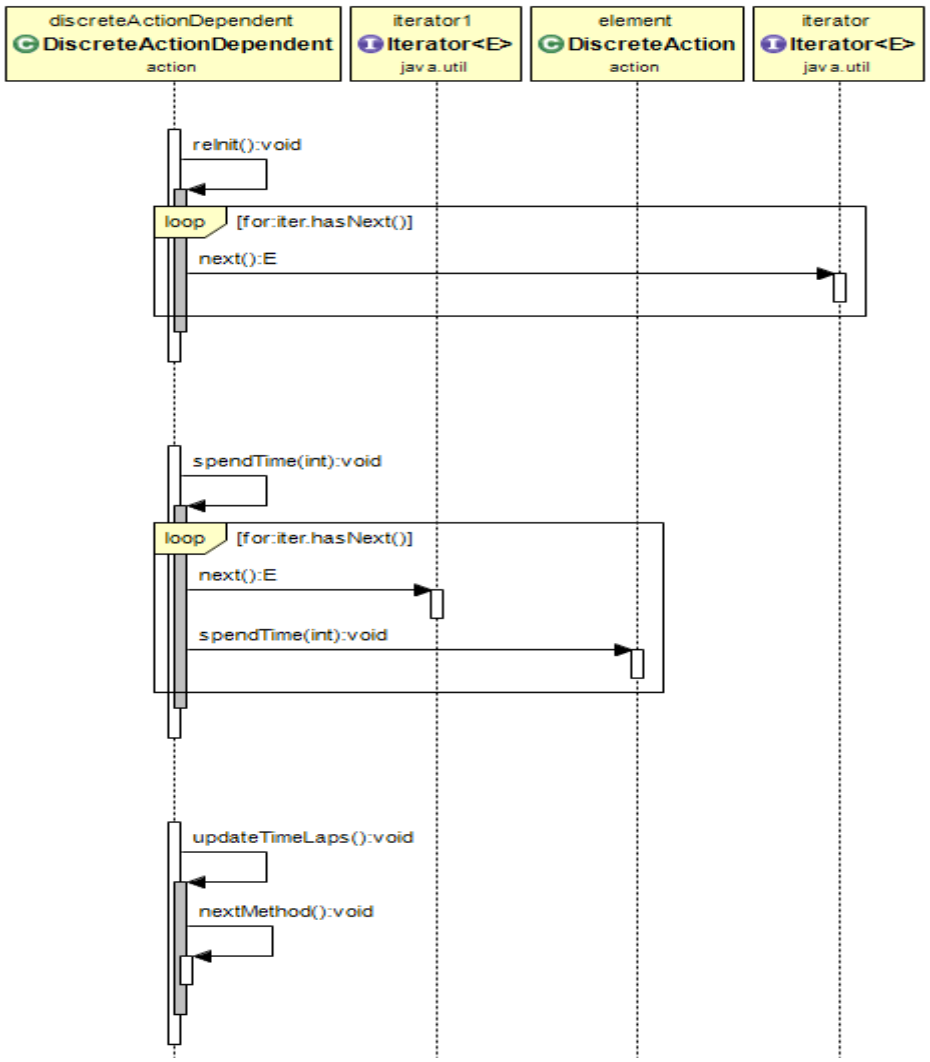


Diagramme séquence : Méthode reInit

Cette méthode ne retourne rien (void). Elle permet d'associer à un DiscretAction element le dernier élément de iter, qui est l'iterator de depedentAction.

Diagramme séquence : Méthode spendTime

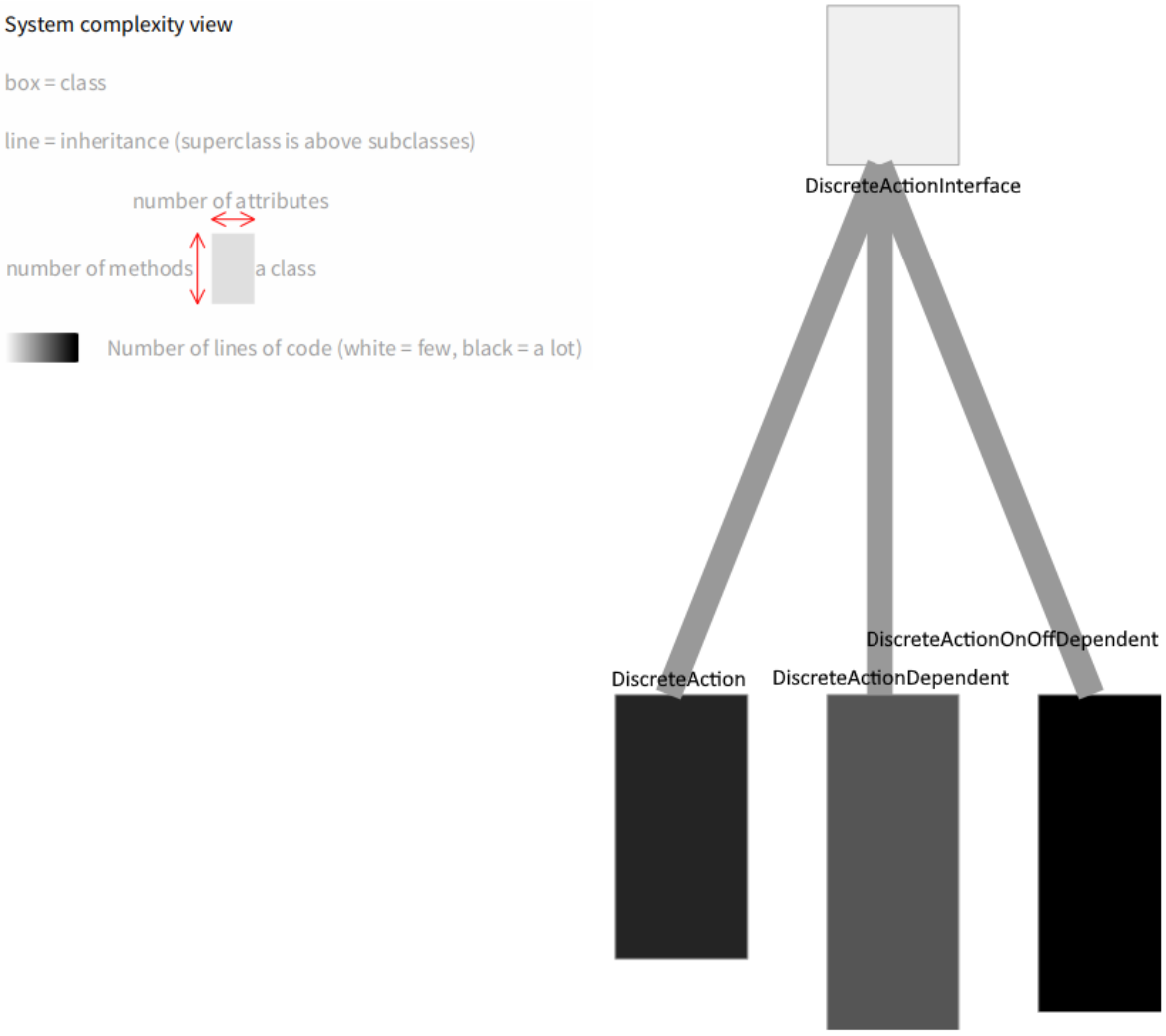
Cette méthode ne retourne rien (void) et elle prend en paramètre un entier t. Elle fait le même for que reInit. A chaque itération, l'élément suivant de iter (qui est l'iterator de depedentAction) est associé à un DiscreteAction element, puis la méthode spendTime de element est appelée avec pour paramètre t. Elle permet de fixer le lapsTime de element à -t si l'ancien lapsTime de element n'était pas nul.

Diagramme séquence : Méthode updateTimeLaps

Cette méthode ne retourne rien. Elle appelle la méthode nextMethod.

Cette classe permet de gérer une série (TreeSet) de DiscreteAction.

Reverse Enginneering avec Moose



Cette visualisation obtenue permet de voir avec Moose l'état du package. Les cases sont les classes (3 dans notre package) et l'interface (1). La largeur de la case représente le nombre d'attributs de la classe / interface. Sa hauteur représente le nombre de méthodes et sa couleur le nombre de ligne de code (sans compter les commentaires, plus c'est noir plus il y a de lignes de code). Il est impossible d'ajouter

une échelle pour avoir une idée de l'ordre de grandeur. Avec cette visualisation, on remarque que DiscreteActionOnOffDependent et DiscreteAction possèdent plus de ligne de code que DiscreteActionDependent alors qu'elles possèdent moins de méthodes. On pourra axer le refactoring sur ces deux classes. Cependant, nous avons essayé de visualiser le nombre de commentaire par classe (DiscreteActionDependent en possède énormément), mais Moose nous montre un rapport de nombre de ligne commenté sur nombre de ligne totale, donc on ne peut pas bien visualiser ce fait (l'interface comprend beaucoup de commentaire par rapport à son nombre total de ligne, donc elle apparaît comme étant la classe la plus commentée…) avec Moose.

Vous trouverez dans le dossier « ReverseEngineering / Action / Moose / System Attraction » un fichier HTML à ouvrir dans votre navigateur. C'est la représentation sous forme de graphe des classes (en noir) et de leurs attributs (en bleu) et méthodes (en rouge) et les relations qu'ils ont avec les autres éléments de la même classe ou d'une autre classe. Il y a un petit bug sur les liens entre la classe DiscreteActionOnOffDependent et l'interface DiscreteActionInterface (normalement la première devrait être reliée à DiscreteAction et non à l'interface, dans le sens où ses méthodes appel des méthodes de DiscreteAction, bien qu'elle implémente DiscreteActionInterface). Cette visualisation nous montre les interactions entre les différentes méthodes, on pourra axer le refactoring sur ces méthodes qui interagissent avec d'autres pour voir s'il est possible de limiter cette interaction.

Vous trouverez dans le dossier « ReverseEngineering / Action / Moose / BluePrint» un fichier HTML à ouvrir dans votre navigateur. L'aide en haut à gauche (point d'interrogation rouge) vous donnera les explications nécessaires pour comprendre cette visualisation. Cette visualisation nous permet de voir que les classes possèdent beaucoup de méthodes publiques, on pourra regarder s'il est possible de les passer en privé lors du refactoring.

Javadoc

Vous trouverez la Javadoc complétée dans le dossier « ReverseEngineering / Action / Javadoc ».