Rapport Debugger

Yuna OTRANTE CHARDONNET & Loreleï NOIRAUD

Utilisation

Une fois le JDISimpleDebugger lancé, il suffit d'entrer les commandes listées ci-dessous (en vert) lorsque le programme affiche “Enter command :”.
Nous conseillons de poser des breakpoints dès le début, sinon le code s'exécutera

Conception

Afin d'introduire ce projet nous avons récupéré le code source prévu à cet effet et nous avons implémenté la commande step du debugger.

Command pattern

Pour commencer, nous avons décidé d'implémenter le design pattern Command dans notre projet afin d'éviter au maximum les différentes vérifications quant aux commandes à exécuter. Ce pattern est basé sur une classe, que nous avons appelé Command qui contient une méthode execute qui renvoie un Objet java ainsi qu'une méthode print.
La méthode execute va donc par la suite faire le travail requis afin que la commande s'exécute correctement.

La méthode print quant à elle se contente d'afficher le résultat s' il y en a un.

Chaque objets “Command” sont stockés dans une map caractérisée par la commande que doit entrer l'utilisateur. Ainsi nous pouvons retrouver les instances facilement.

Appel de commande

Nous demandons à l'utilisateur d'entrer une commande dans la boucle qui lit les évènements de la VM du debuggee à condition que l'événement ClassPrepareEvent soit terminé (afin d'attendre la fin d'initialisation du projet). Cette solution n'est pas la meilleure option mais c'est celle que nous avons choisie afin que nous puissions exécuter une commande à chaque évènement. Notamment le step que nous pouvons changer de type (step/step-over) en cours de route grâce à l'enregistrement de l'événement Breakpoint dont dépend la StepRequest en cours.

Commandes implémentées

Step-over : step-over

La méthode step over est très similaire à la méthode step réalisé dans l'introduction. Nous réalisons les mêmes vérifications, mais les paramètres lors de la création de la requête de step sont différents : **StepRequest.STEP_LINE**, **StepRequest.STEP_OVER**.

- **STEP_LINE** permet d'avancer de ligne en ligne lors du step.
- **STEP_OVER** permet de ne pas entrer dans chaques méthodes sur lesquelles nous passons.

Ainsi notre step est bien paramétré et permet de passer à la prochaine ligne sans entrer dans les méthodes appelées.

Continue : continue

La méthode continue a été faite de manière à supprimer toutes les possibles **StepRequest** enregistré dans la VM du debuggee enregistré dans notre debugger. Ainsi le step actif n'est plus et le prochain évènement est un **BreakpointEvent** ou un **VMDeathEvent**.

Frame : frame

La commande frame récupère la frame actuelle du debugger. Nous la récupérons grâce à la VM enregistrer dans notre debugger. Nous l'imprimons via la méthode **print**() afin de visualiser ses attributs, notamment la localisation (le nom de la classe ainsi que la ligne), ainsi que les variables présentes dans cette classe. Exemple :

```
frame
Location : dbg.JDISimpleDebuggee:7
Variables :
name : args
type name : java.lang.String[]
value : instance of java.lang.String[0] (id=469)

name : description
type name : java.lang.String
value : "Simple power printer"
```

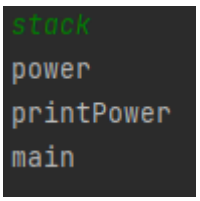
Temporaries : temporaries

Pour cette commande nous avons pu récupérer la frame grâce à la commande précédente et récupérer uniquement les variables que celle-ci contient. Ainsi grâce à la méthode print nous avons fait en sorte que les variables soient affichés sous la forme
nom -> valeur.

Stack : stack

La méthode stack permet de récupérer la liste des méthodes appelées jusqu'ici. Nous avons remarqué qu'à chaque appel de méthode, une nouvelle frame est ajoutée dans notre VM.

Nous avons créé une liste contenant tous les noms des méthodes où chaque frame se trouve. Exemple :



Receiver : receiver

Pour la commande receiver nous avons mis du temps à comprendre son objectif. Celui-ci étant de renvoyer la classe objet (this) qui a reçoit la méthode courante. Nous avons mis beaucoup de temps à comprendre que lorsqu'une méthode est static, alors celle-ci n'a pas de receveur. Il est donc normal que la méthode **thisObject()** renvoie null dans ce contexte.

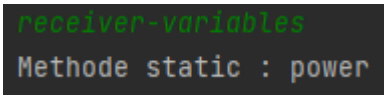
Sender : sender

De la même manière que la commande précédente nous avions mis ces fonctionnalités de côté. Mais une fois comprises, ces méthodes ont été rapides à implémenter. Sender est donc le **thisObject** de la frame ayant appellé la frame courante, de la même manière, le résultat peut renvoyer null.

Receiver-variables : receiver-variables

Cette méthode permet d'afficher les variables du receiver. Nous appelons donc receiver et sauvegardons ses variables sous la forme d'une liste de Value que nous pouvons donc print par la suite sous la forme nom -> valeur.

Lorsque le résultat est null alors on affiche une "alerte" précisant cette situation. Exemple :



Method : method

La commande méthode appelle la commande frame puis renvoie la méthode contenue dans la location de la frame. Cette chaîne de caractère pourra ensuite être imprimée.

Arguments : arguments

Arguments permet de renvoyer les arguments de la méthode en cours d'exécution. Il s'agit donc d'un **Map** avec comme clé le nom et comme valeur la Value. Nous les récupérons grâce à la méthode contenu dans la frame. Ainsi nous pouvons les récupérer et les afficher sous la forme nom -> valeur.

Print-var : print-var(varName)

La commande print-var permet d' envoyer la valeur d'une variable entrée en paramètre, pour ce faire il faut déjà analyser la chaîne de caractère entrée par l'utilisateur afin de déterminer la commande ainsi que le paramètre. Une fois le nom de la variable récupéré, nous pouvons appeler la commande frame et appeler sur son résultat la méthode :

visibleVariableByName. Nous enregistrons le nom de la variable enregistrée puis nous l'affichons via la méthode print de la Command.

Break : break(filename, lineNumber)

La méthode break permet de placer des breakpoints dans un certain fichier a une certaine ligne. Pour ce faire, une fois toute la chaîne de caractère parser, nous devons trouver l'objet class qui correspond à notre fichier entrer en paramètre pour pouvoir ensuite faire une **BreakPointRequest** qui permettra de poser le breakpoint.

Breakpoints : breakpoints

Cette commande permet de lister tous les breakpoints qui sont encore actifs. Nous pouvons récupérer la liste via la VM et le **eventRequestManager**.

Break-once : break-once(filename, lineNumber)

Cette commande permet au breakpoint de se désactiver une fois atteint. Il s'agit juste d'exécuter la commande break sauf que lorsqu'un événement breakpoint apparaît nous vérifions si l'événement actuel correspond à un événement breakpoint qui est contenu dans une liste des breakpoints once que nous avons créer. Dans ce cas, ce breakpoint est supprimé de nos requêtes.

Break-on-count : break-on-count(filename,lineNumber,count)

Le break-on-count permet d'activer un breakpoint après un certain nombre de passages sur celui-ci. Pour ce faire nous avons créé une map avec pour clé le Breakpoint et comme clé un compteur. Dans le cas où nous passons sur un breakpoint et que celui-ci est compris dans la map alors on décrémente son compteur. Quand le compteur est à 0, le Breakpoint est réellement créé.

Break-before-method-call : break-before-method-call(methodName)

Pour cette commande, on fourni le nom de la méthode en paramètre. Cette méthode permet de placer un breakpoint juste avant la méthode concernée. Pour ce faire, on retrouve la méthode dans la classe que nous sommes en train de debugger et on place le breakpoint à la première ligne de la méthode grâce à la commande break.

IHM

Une fois toute ces commandes implémenter nous avons décider de mettre en place une IHM. Malheureusement le lien entre notre "back" et l'IHM pose certains problèmes dû à l'architecture actuelle de notre back.