ACO : Rapport du projet mini-éditeur

LE FLEM Erwan
LEBRETON Mickaël

Groupe M1 SSR
Encadrés par Firas Hmida

07/12/2017

Diagrammes de classes

Les images sont beaucoup trop grandes pour passer sur une feuille A4. Nous vous invitons à vous référer aux fichiers suivants :

- ./diagrammes/v3/receiver_v3.jpg
- ./diagrammes/v3/invoker_v3.jpg
- ./diagrammes/v3/memento_v3.jpg
- ./diagrammes/v3/command_v3.jpg

Choix d'implémentation

La gestion des commandes.

`CommandEnum` est juste une énumération qui décrit la liste des commandes disponible sans exposer d'implémentation particulière.

L'IHM n'a de dépendance que vers l'énumération des commandes, le contrôleur et rien d'autre.
Pour les commandes, le contrôleur n'appelle jamais directement les opérations du moteur tel que sélectionner copier etc... Il faut passer par l'interface `Command`.

Le contrôleur a en mémoire une table de hashage qui à une clé de type `CommandEnum` associée à une instance de `Command`. Cette map est donnée au constructeur.
C'est dans le main où l'on crée les associations d'une `CommandEnum` à une implementation de l'interface `Command`.
Le main est la seul classe où on utilise explicitement les implémentations de commande (pour leur constructions).
L'intérêt est qu'il est très facile de changer d'implémentation ou d'ajouter des commandes sans changer ni l'IHM ni le contrôleur. Le contrôleur ne connaît pas de commande particulière, il ne voit que l'interface `Command`.

L'interface `ReplayableCommand`

Cette interface est une forme de memento. Elle correspond en fait à la V2 du projet.

Elle permet de rejouer des commandes : à chaque fois que l'on exécute une commande, si l'enregistrement est activée on la signale à l'enregistreur.

On le fait pour toute les commandes, puisque le contrôler n'a pas conscience de quelle commande il exécute : Il ne voit que l'interface commande.

Mais on a rencontré un problème, toutes les commandes n'ont pas de sens d'être rejouées, c'est par exemple le cas de la commande de démarrage de l'enregistreur lui même ou encore des commandes `undo`, `redo`. Même remarque pour les memento, on n'annule pas un `undo` en lui même.

Parmi les solutions envisagées mais non retenues :

- Faire que les méthodes `getReplayableCommand` et `getMemento` retournent `null` si elles ne peuvent être contenu dans une macro.

Problème : C'est une mauvaise solution, car cela force l'utilisation d'une conditionnelle supplémentaire dans plusieurs partie du code. Généralement prendre en paramètre null ou renvoyer null est considéré comme une mauvaise pratique.

- Pour les mémentos, faire que les commandes rejouables (exemple `couper`) passe par une `Command` mais que les non rejouables (exemple `copier`) passent directement par le moteur. Cela évite d'avoir à renvoyer null.

  Problème : C'est une mauvaise solution car on force deux modes d'utilisations pour une action sensée être générique.
  Si on décide qu'une commande à la base non-rejouable peut finalement avoir du sens d'être rejoué, on doit changer pas mal de code.
  Il est plus facile de provoquer des bugs quand il y a plusieurs moyen différents mais incompatibles entre eux que d'exécuter une commande.

- Faire un `switch case` sur le type de la commande (en utilisant `instanceof` ou `getClass()`) pour décider de la manière de la gérer.

  Problème : C'est probablement la pire des solutions car elle force l'utilisation différenciée de commandes sensées être générique. On perd tout l'intérêt du polymorphisme. De plus la classe qui fait usage d'`instanceof`/`getClass()` doit avoir connaissance des classes concrètes au lieu de juste voir une interface abstraite.

La solution retenu est d'utiliser un pattern appelé "Objet spécial". Par exemple si dans une application de gestion réseau vous voulez représenter qu'une machine peut ne pas avoir d'ip, 0.0.0.0 /0 est un *objet spéciale* plus adapté que l'utilisation de `null`.
Dans notre cas, l'objet spécial est une instance de ReplayableCommand qui ne fait rien ou bien un Memento vide.

Le moteur

Le moteur est le coeur de l'application, il contient l'état du moteur (contenu, sélection en cours, presse papier) et offre des méthodes pour agir dessus et effectuer concrètement les commandes.

Toutes les opérations du moteur on lieu sur la sélection actuelle définie par l'appel à `selectionner(Selection s)`.

Les méthodes ne prennent donc pas en argument leur lieu d'exécution, il est défini avec le dernier appel à `selectionner(Selection s)`.

On sépare ainsi la gestion des préoccupations et on centralise au même endroit la gestion de la sélection.
Cela rend la création des commandes et des mémentos plus simple. Sans classe `Selection` et si on utilisait des types primitifs comme argument aux commandes copier et autre, cela rendrait l'évolution moins simple (par exemple si l'on veut passer d'une sélection de type "début taille" à une sélection de type "debut fin").

Cette implémentation favorise aussi l'écriture des commandes rejouables dans les macros : pour les commandes qui n'ont besoin d'aucun argument, la commande rejouable est identique à la commande en elle même. Par exemple les classes copier et coller implémentent à la fois l'interface `Command` et l'interface `ReplayableCommand`.

Nous avons aussi pris la décision de faire en sorte que les instances de `SelectionImpl` soient immuables, c'est à dire non modifiable, une fois construite il est impossible de modifier leur état. Rendre les sélections immuables empêche notamment au client du moteur de faire des sélections illégales.
Voici un exemple de ce qui pourrait se passer sans immuabilité avec des setters :

```
Selection s = new Selection(10, 5);
m.selectionner(s)
s.setSize(10000000);
```

Puisqu'en Java on fonctionne avec des valeurs de références, l'attribut de sélection du moteur pointe vers le même objet que la référence s dans le client. Sans recopie défensive (construire une nouvelle sélection interne avec le même état), on peut donc modifier de l'extérieur l'état de le sélection sans contrôle du moteur et ainsi contourner la précondition dans `selectionner(Selection s)`.

Avec une classe immuable, on évite ce problème sans avoir besoin de faire de recopie défensive et le risque de bug est moindre.

C'est un exemple de *"preamature optimization is the root of all evil"* : on pourrait au départ vouloir faire une classe sélection muable pour éviter la création de plein de petits objets, mais cela serait rendu inutile dès le moment où l'on met en œuvre la recopie défensive dans `selectionner(Selection s)`.

Les mémento sont eux aussi immuables.

Rapport de test

Pendant l'écriture des tests, il fallait penser à gérér les différentes classes de cas possibles ainsi que les cas limites :

- Sélection au tout début, au milieu mais aussi à la fin, pour vérifier qu'il n'y a pas de bug de type "décalage de 1".
- Test des commandes à la fois avec une sélection vide et une sélection non vide.
- Pour la gestion de `undo` et `redo` : Faire des tests avec un `undo` un `redo`, plusieurs `undo` plusieurs `redo`, faire des tests avec des imbrications possibles.

Pour nous assister lors des tests unitaires, nous avons placé quelques assertions. Dans la classe `Moteur` cela nous a permis de trouver des bugs provoqués par l'oubli de cas limites. Par exemple nous avons un texte de 20 caractères. Nous sélectionnons les caractères 15 à 20, ce qui est valide puis effectuons une commande coller alors que le presse papier ne fait qu'un caractère. On se retrouve donc avec une sélection qui va de 15 à 20 alors que la taille du texte est maintenant de 16. Nous avons donc un moteur avec une sélection courante invalide.

L'assertion nous a permis la détection de ce genre de bug, et à permis l'écriture de cas de tests oubliés avant.

- Lors des tests unitaires, on peut vouloir utiliser et tester des commandes.
- Problèmes : certaines commandes, tel que sélectionner ou inserer, demandent à l'utilisateur
- une intervention pour choisir des paramètres.
- Pour rappel on avait fait ça dans le but d'éviter à l'ihm de devoir connaitre en avance les arguments d'une commande, pas compatible avec du polymorphisme sur une méthode très générique.
-
- On pourrait lors des tests unitaires utiliser une vraie implémentation d'ihm mais cela perdrait de son intérêt puisque ces tests ne serait plus automatisés.
-
- La solution classique à ce problème est utiliser une doublure de test.
- Cette doublure permet d'imiter le comportement d'une vraie IHM, comme si l'utilisateur avait lui même entré la réponse.