CmdLineParser

Le compte-rendu des quatre séances

Exercices 1 à 2

Quels problèmes posent le code proposé de l'exercice 1 ?

Le problème avec le code proposé à l'exercice 1 est que les instructions permettant de vérifier la présence des options et de l'exécution des actions conséquentes sont réalisées dans le main de la classe Application alors que cette partie du code devraient être réaliser par la méthode process de la classe CmdLineParser. Pour pallier ce problème, nous avons utilisé le design pattern Strategy que nous avons étudié en cours.

Quel doit être le type de la méthode registerWithParameter ?

Cette méthode prend en paramètre le nom de l'option ainsi que le code a exécuté sous la forme d'un consumer d'iterator de string. Cette méthode ne renvoie rien.

Que doit-on faire quand le paramètre d'une option n'est pas présent lors de l'appel à process ?

Lorsque le paramètre d'une option n'est pas présent lors de l'appel à process, nous devons lever une Parse Exception.

Quel est le type que l'on doit stocker dans cette map unique ?

Cette map doit associer à chaque nom d'option, un consumer d'iterator de string. L'idée est de fournir aux codes, l'itérateur de string afin qu'il puisse y récupérer les arguments dont ils ont besoin pour réaliser leurs tâches. Dans le cas des flags, on peut ajouter le runnable sous la forme d'un iterator de string à la map, à l'aide d'une lambda en indiquant que le paramètre - l'iterator - n'est pas nécessaire et que sa tâche est l'exécution du runnable.

En plus des questions posées dans le sujet, précisez quel(s) accès avez vous laissé (pour le main de Application) à la classe CmdLineParser ?

Désormais, le main de la classe Application n'a accès qu'aux méthodes : addFlag, addOptionWithOneParameter et process de la classe CmdLineParser.

Quels sont les modificateurs d'accessibilité des champs et des méthodes dans cette dernière ?

La seule propriété de la classe CmdLineParser est de type map, se nomme registered Options et est privée. Les trois méthodes que nous venons de citer à la question précédente - et unique méthode de la classe - sont toutes les trois publiques.

Comment gérez-vous la situation où une option est enregistrée plusieurs fois (plusieurs appels à la méthode addFlag() avec le même nom) ?

Lors de l'ajout d'un flag, on vérifie qu'un flag ou une option n'existe pas déjà pour son nom, si c'est le cas on lève une IllegalStateException.

Comment avez vous représenté l'action à réaliser lorsqu'on rencontre une option enregistrée par addFlag() ?

L'action à réaliser lorsque l'on rencontre une option enregistrée par addFlag est un runnable que nous enveloppons dans un consumer d'iterator de string comme expliqué précédemment.

Même question pour une option enregistrée avec addOptionWithOneParameter() ?

L'action à réaliser est représentée sous la forme d'un consumer d'iterator de string.

Comment les avez vous représentées de manière cohérente dans une seule et même map ?

Comme nous l'avons déjà expliqué précédemment, la map prend un consumer d'iterator de string - ce qui correspond aux options avec paramètres - et gère les flags en créant un consumer d'iterator de string qui se fiche de l'itérateur en entrée et qui se contente d'exécuter la méthode run du runnable du flag.

Décrivez comment l'exécution d'une telle action a accès à son argument, le cas échéant ?

L'action prend en paramètre l'itérateur de string, il peut donc prendre le prochain paramètre de l'itérateur s'il en a besoin pour sa tâche. Dans le cas où l'argument dont il a besoin n'est pas présent, il lève une NoSuchElementException qui sera capté par la méthode process qui la transformera par la suite en une ParseException afin d'indiquer l'erreur.

Cet accès est-il sécurisé? protégé? pose-t-il des problèmes ?

Oui, c'est accès pose des problèmes car les lambdas ont un total accès à l'iterator et peuvent par conséquent le modifier comme bon leur semble. Cet accès est donc non sécurisé et non protégé.

Comment gérez-vous le cas où il manque un paramètre ?

Comme expliqué ci-dessus, lorsqu'il manque un paramètre on lève une exception afin de signaler l'erreur à la méthode process.

Le cas où il y a une erreur de conversion d'un paramètre de String vers int ?

Dans le cas d'une erreur de conversion, on lève une ParseException de la même manière que pour le manque d'argument.

Listez les cas/situations que vous avez testé avec vos tests unitaires (en langage naturel, de manière synthétique).

Les situations couvertes par les tests unitaires :

- Pour vérifier l'activation d'un flag
- Pour vérifier l'activation d'une option avec paramètre
- Pour vérifier que le parsing d'une option qui nécessite un paramètre - mais qui n'en trouve pas - lève une erreur
- Pour vérifier qu'on ne peut pas ajouter deux options avec le même nom

Exercice 3

Quel design pattern permet de séparer la responsabilité de création des objets de la classe PaintOptions (ou PaintSettings) avec ses multiples paramètres possibles et permet de fournir des valeurs par défaut ?

C'est le design pattern builder.

Listez les avantages procurés par le design pattern que vous avez mis en œuvre dans l'exercice 3 ?

Le design pattern builder est utilisé pour la création d'objets complexes dont les différentes parties doivent être créées suivant un certain ordre ou un algorithme spécifique. Une classe externe contrôle l'algorithme de construction. Comme expliqué dans le cours, ce design pattern permet de rendre la construction des objets complexes plus simple à prendre en main et surtout plus lisibles; notamment pour les objets qui nécessite beaucoup d'arguments car le design pattern permet de les ajouter individuellement. Le design pattern permet également de prendre en charge la vérification des arguments optionnels afin d'assurer que l'objet créée est conforme aux règles de gestion - vérifications qui doivent être réalisées par d'autres fonctions sans l'utilisation de ce modèle. Enfin le fait de déléguer le processus de création des propriétés de l'objet à une autre classe permet d'empêcher les développeurs de lancer des processus depuis l'intérieur du constructeur de l'objet, ce qui assure par conséquent les bonnes pratiques.

Exercices 4 à 5

Pourquoi est-il devenu important de créer une classe Option dès l'exercice 4 ?

Il est important de créer une classe Option dès l'exercice 4 car dans le cas contraire, il aurait fallu créer une nouvelle Map pour chaque nouvelle propriété que nous souhaitons ajouter à l'option.

Discutez de ce que vous avez mis comme responsabilités dans cette classe dans l'exercice 4.

Dans l'exercice 4, nous avons donné comme propriété à la classe Option : un nom, un consumer d'arguments, un nombre d'arguments ainsi qu'une information indiquant si l'option est obligatoire. La classe est donc responsable de la connaissance des propriétés

qui lui sont liées. La création de l'objet est toujours déléguée au constructeur OptionBuilder qui est une classe interne de Option - on peut donc dire qu'elle est responsable de la vérification des propriétés obligatoires.

Discutez de ce que vous avez mis comme responsabilités dans cette classe dans l'exercice 5.

Dans l'exercice 5, nous avons ajouté à la classe Option la responsabilité de connaître ses alias, les options avec lesquelles elles sont en conflits ainsi que sa documentation.

Discutez de tous les moyens offerts aux utilisateurs (depuis le main d' Application) pour en créer des instances dans les exercices 4 et 5.

Dans les exercices 4 et 5, les utilisateurs peuvent créer des instances d'option à l'aide de l'option builder. L'utilisateur doit alors renseigner le nom, le nombre d'argument et le consumer de liste de string de l'option et peuvent par la suite préciser l'instance en utilisant les mutateurs du builder. Lorsque l'utilisateur a renseigné les propriétés qu'il souhaitait modifier, il peut essayer de demander une instanciation de l'objet avec la méthode build, ce qui fonctionne si tous les champs obligatoires ont été remplis.

Discutez de la manière dont vous avez assuré la compatibilité ascendante entre le code de la version 3.0 et celui de la version 2.0.

La première tâche a été de transformer la map qui associait au nom de l'option le consumer en une map qui associe au nom de l'option, l'option. Lors de l'ajout d'un flag, il faut désormais ajouter une option en précisant 0 comme nombre d'argument, le nom et le consumer comme réalisé précédemment. Il faut ensuite modifier la méthode addOptionWithOneParameter qui prend désormais une option en paramètre.

La seconde tâche concerne la méthode process, maintenant que nous avons connaissance du nombre d'arguments, on peut les récupérer dans une liste - en vérifiant au passage que se sont bien des arguments - et les transmettre au consumer.

Listez les cas/situations que vous avez testé avec vos tests unitaires (en langage naturel, de manière synthétique).

Les situations couvertes par les tests unitaires :

- Afin de vérifier qu'il est impossible de parser quelque chose de null.
- Afin de vérifier que le code d'une option est correctement exécutée.
- Afin de vérifier que la présence d'un flag est bien prise en compte.
- Afin de vérifier qu'une option avec 1 paramètre est bien prise en compte.
- Afin de vérifier qu'une option avec 2 paramètres est bien prise en compte.
- Afin de vérifier qu'une option ne peut pas prendre comme argument quelque chose qui pourrait spécifier une autre option.
- Afin de vérifier que le processus échoue si une option obligatoire n'est pas présente.

La gestion de la "consommation" des paramètres par l'exécution de l'action associée à une option lorsqu'elle est rencontrée sur la ligne de commande par process() a-t-elle changé depuis la fin de l'exercice 2? Si oui, décrivez pourquoi et comment sont gérés les problèmes (du genre paramètre manquant ou incorrect)?

Comme nous venons de l'expliquer, désormais nous connaissons le nombre d'arguments utilisé par l'action, ce qui n'était pas le cas dans l'exercice 2, donc oui la gestion de la consommation est très différente. Désormais nous pouvons itérer sur les arguments dans la méthode process en fonction des besoins des options - y vérifier directement si le nombre d'options est bien présent - et envoyer uniquement à l'action les arguments dont il a besoin. Lorsqu'une erreur est rencontrée, une ParseException est levée comme précédemment.

Décrire brièvement comment vous avez fait (à quel endroit dans le code et dans vos classes) pour gérer les alias.

Nous avons gérer les alias dans la méthode addOptionWithOneParameter. Dans cette méthode, à chaque fois que nous ajoutons une option au dictionnaire, nous ajoutons également les alias au dictionnaire avec comme valeur: la même option.

Décrire brièvement comment vous avez fait (à quel endroit dans le code et dans vos classes) pour gérer la documentation.

La classe CmdLineParser possède une méthode publique nommée usage qui permet de construire le message d'aide. La méthode usage itère sur les options enregistrées - de manière triée par ordre lexicographique - et ajoute pour chacune sa documentation à l'aide d'un String. Lors de l'enregistrement de l'option help, on indique comme action d'effectuer un affichage de la chaîne retournée par la méthode usage.

Décrire brièvement comment vous avez fait (à quel endroit dans le code et dans vos classes) pour gérer les conflits entre options.

Nous avons géré les conflits entre options à la fin de la méthode process, nous vérifions pour chaque option si c'est des options en conflits sont présentes. Dans le cas où l'événement se produit nous levons une ParseException.

Quels problèmes voyez-vous dans la solution que vous avez mise en œuvre à la fin de l'exercice 5 ?

Je pense que nous commençons à donner trop de responsabilités à la méthode process, ce qui a pour conséquence de rendre la maintenabilité du code complexe. Nous devrions envisager de créer des observateurs pour réaliser ces tâches.

Exercices 6

Les observers qui sont utilisés dans l'exercice 6 sont-ils *push* ou *pull* ?

Les observateurs qui sont utilisés dans l'exercice 6 sont push car nous transmettons les options aux méthodes des observateurs au moment de la notification.

Quels sont les avantages et inconvénients d'utiliser le patron *observer* pour traiter les relations entre les options ?

Les avantages de l'utilisation de ce patron est le gain en maintenabilité. Comme expliqué durant le cours, si jamais une nouvelle fonctionnalité devait être ajoutée à l'application, il suffirait alors d'ajouter un observer et la mission serait réalisée - et ce sans avoir complexifié le code existant.

Le désavantage que je trouve à l'utilisation de ce patron - mais c'est probablement que je l'ai mal utilisé - c'est que je suis obligé de stocker des informations complémentaires dans les observers pour réaliser leurs tâches, ce qui fait de la duplication d'informations.

Dans votre code, quelle est l'utilité que les méthodes de l'interface OptionsObserver prennent l'OptionManager en paramètre.

Je n'ai pas compris en quoi les observers avaient besoin du manager, je n'avais pas fait attention aux signatures des méthodes des observers durant les TP et maintenant il est trop tard pour vous poser la question.

Donnez un exemple précis où le code d'un de vos OptionsOberser utilise l'OptionManager.

Je n'en ai aucun qui utilise le manager.

Les conflits entre options peuvent être déclarés au niveau de l'option ou au niveau de CmdLineParser. De plus, le conflit peut être déclaré avec le nom de l'option ou avec un objet Option. Quelle solution avez vous choisi ?

La gestion des conflits est réalisée par l'observer et se base sur les noms des options.

Discutez des avantages et inconvénients de chaque choix.

Nous avons défini la comparaison des options par leur noms mais je pense que définir la comparaison sur l'objet aurait pu être un autre choix possible, voire même peut être plus précis que par simplement le nom. La méthode sur le nom de l'option est peut être moins représentative de l'objet mais elle a l'avantage d'être plus souple et plus simple à être implémentée.

Au niveau de l'emplacement de la déclaration des conflits, l'avantage de déclarer les options en conflits au niveau de l'option est - je trouve - plus cohérent car c'est une propriété de l'option. Déclarer une donnée propre à une option dans une classe externe est peut être plus difficilement maintenable. Je dirais que l'avantage de cette méthode est de pouvoir manipuler directement les données depuis la classe CmdLineParser et de pouvoir les modifier à tout moment.

Implicitement on suppose que la méthode process n'est appelée qu'une seule fois. Quelles modifications faudrait-il faire pour pouvoir appeler process plusieurs fois (soyez précis) ?

Si on souhaite pouvoir utiliser la méthode process après sa première exécution, il faudrait obligatoirement réinitialiser les propriétés des objets ainsi que les propriétés des observers dont les états ne se réinitialiseraient pas automatiquement avant le début du deuxième appel.

Exercices 7

Si l'on voulait implémenter une stratégie SMART RELAXED similaire à RELAXED mais qui s'arrête à la première option déclarée auprès du CmdLineParser et pas simplement qui commence par un tiret. Est-ce que cela est possible avec l'interface ParametersRetrievalStrategy ?

Non parce qu'il faudrait avoir accès à la méthode process option de l'option manager, ce qui n'est pas possible actuellement. Pour pallier ce problème, il faudrait ajouter l'option manager en paramètre de l'interface fonctionnelle, ce qui signifie que nous n'avons plus une BiFunction mais désormais une TriFunction avec : l'iterator, l'option et l'option manager.

Décrivez les modifications à apporter à l'interface ParametersRetrievalStrategy et donnez le code de la strategy SMARTRELAXED.

Il faut donc changer la signature de l'interface fonctionnelle qui passe de :

BiFunction<Iterator<String>, Option, List<String>>

A une :

TriFunction<Iterator<String>, Option, OptionsManager, List<String>>

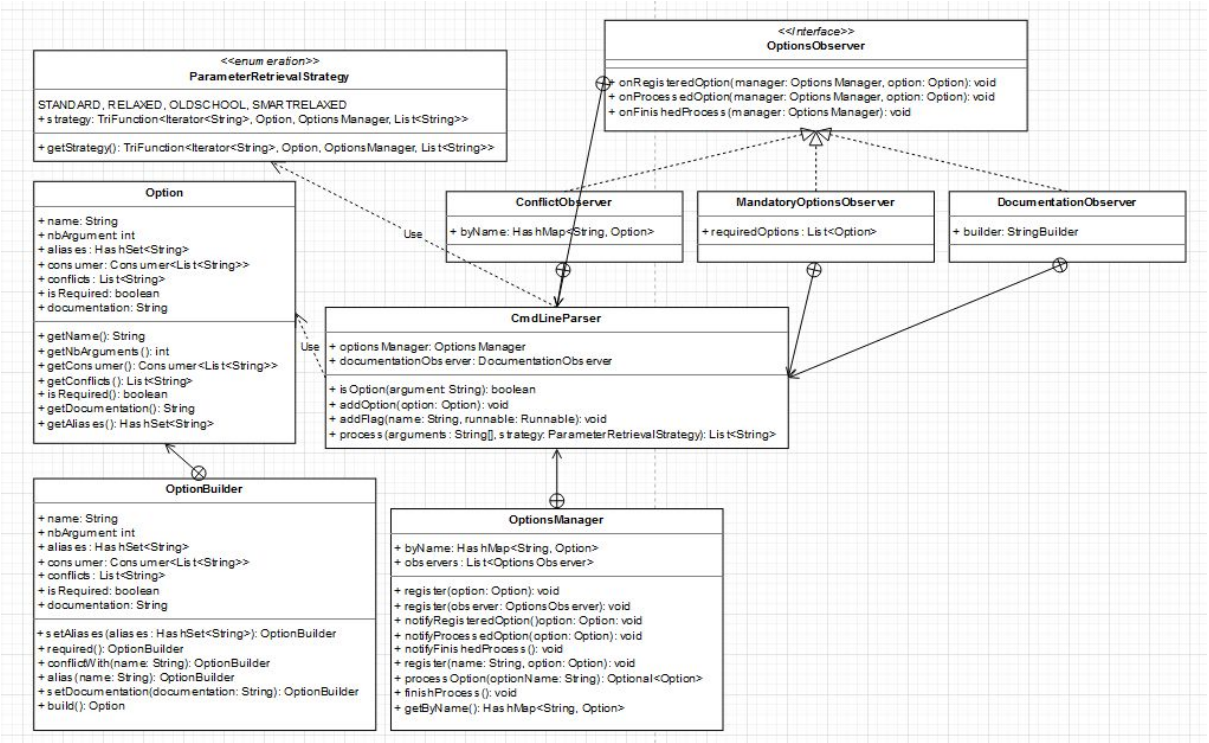
Ensuite il faut modifier la condition d'arrêt en regardant si l'optional renvoyée par process option est vide, et dans ce cas, arrêter.

```
SMARTRELAXED((iterator, option, optionManager) -> {
    List<String> list = new ArrayList<>();
    for (int i = 0; i < option.getNbArguments(); i++) {
        String parameter = iterator.next();
        if (!optionManager.processOption(parameter).isEmpty()) {
            break;
        }
        list.add(parameter);
    }
    return list;
});

private final TriFunction<Iterator<String>, Option, OptionsManager, List<String>> strategy;
```

Décrivez l'architecture globale de la librairie CmdLineParser avec un texte et un diagramme UML.

Le diagramme UML :



Comme vous pouvez le voir sur l'architecture décrite ci-dessus, nous avons utilisé de nombreux design pattern pour la réalisation de ce projet. Nous avons commencé avec le patron Builder afin de pouvoir créer plus aisément nos options qui devenaient des objets complexes avec des propriétés optionnelles. Nous avons ensuite utilisé le design pattern observer pour gagner en maintenabilité et en lisibilité pour les traitements à réaliser sur les options. Ce sont les observers ConflictObserver, MandatoryObserver et DocumentationObserver qui s'occupent respectivement de la gestion des conflit, des options obligatoires et de la génération de la documentation. Enfin nous retrouvons l'utilisation du design pattern Strategy a de nombreuses reprises durant le projet, que ce soit simplement dans les actions réalisées par les options ou encore dans l'utilisations des différentes Strategy : Standard, Relaxed, Oldschool et Smartrelaxed.