Rapport UGE CmdLineParser

Table des matières

Exercice 1 à 2

- ## Plusieurs appels à la méthode addFlag avec le même nom

Lorsque l'on tente d'appeler la méthode **addFlag**, la méthode **register** de la classe **optionManager** qui a pour but d'enregistrer l'option dans la structure de donnée contenant toutes les options enregistré, regarde si le nom ne correspond pas déjà à une option présente. Si c'est le cas alors on renvoie un **IllegalStateException** précisant que une option de ce nom existe déjà.

- ## Type de l'action enregistrée

Pour la méthode **addFlag** le type de l'action enregistrée est une un **Runnable** car ce sont des options sans paramètre.

Pour la méthode **addOptionWithOneParameter** c'est un **Consumer<String>** car comme l'indique le nom de la méthode, ces options la prennent un seul paramètre, ce qui correspond bien au type Consumer qui prend un paramètre et ne renvoie rien.

La **map** a pour clé le nom de l'option et pour valeur la classe Option. Les actions que ce soit un **Runnable** ou bien un Consumer, sont stocké via un **Consumer<List<String>>** au moment de construire l'option.

Lors de l'exécution d'une action, on a accès aux arguments des options via un **iterator** qui nous permet d'itérer le nombre de paramètre de l'option fois tout en vérifiant à chaque fois si le paramètre est correct c'est-à-dire par exemple que le paramètre ne commence pas par un '-'. Lorsqu'il manque des paramètres à une option une **ParseException** est renvoyé. Avant de parser une String vers un **int** on vérifie d'abord que la String en question contient bien seulement des chiffres au quel cas on renvoie un **IllegalArgumentException** si ce n'est pas le cas.

Les accès aux arguments d'une option peuvent poser problème lorsqu'il y a pas le bon nombre d'arguments attendu par exemple. Cependant ce genre de cas sont gérer afin que le programme ne plante pas.

Test effectué

Test effectué :

- Le cas ou il n’y a aucune option.

- Vérifie que les fichiers retourné par le process correspondent bien à ceux attendu.

- Vérifie qu’une exception est levé lorsque une option obligatoire n’est pas donnée.

- Vérifie que lorsque l’on fait parser un String qui ne contient pas que des chiffre ça nous renvoie bien une exception.

Exercice 3 à 5

Design pattern de l’exercice 3

Dans l’exercice 3 nous avions implémenté le design pattern Builder il nous a permit de :

1. Chaîné nos méthode afin de construire notre option.
2. Choisir les paramètres de l’option car certains ne sont pas obligatoire.
3. Possibilité de fournir des paramètres par défaut.
4. Seul moyens de construire nos option (sécurisé).

Classe Option

La création de la classe Option nous a permit de pouvoir stocker les différents paramètres d’une option qui devenaient de plus en plus nombreux au fur et à mesure de l’avancement de la librairie.

Cela facilite d’autant plus la création d’une option par l’utilisateur qui depuis le main a 3 choix de création d’option parmi les méthodes de **cmdParser** suivante :

- **addFlag** → permet d’enregistrer une option basique c’est-à-dire celle qui n’ont aucun paramètre.

- **addOptionWithOneParameter** → permet d’enregistrer une option avec un seul paramètre tel que **-border-width**.

- **addOption** → permet d'enregistrer une option avec un nombre de paramètre quelconque ainsi que de saisir différent paramètres d'une option grâce au **patron Builder** implémenté pour les Options.

La compatibilité entre la version 2.0 et 3.0

Lors du passage de la version de notre librairie de 2.0 à 3.0 nous avions du garder les méthodes **addFlag** et **addOptionWithOneParameter** afin de s'assurer que le code de la version 2.0 compilerai avec celle du code de 3.0.

Gestion de la consommation des paramètres

Oui la consommation des paramètres d'une option a changé depuis la fin de l'exercice 2 car nous avions du accepter des options avec des arguments. Dans la méthode process, le tableau d'arguments correspondant aux options et leurs arguments si ils en ont est transformé en un **iterator** afin d'avoir une gestion plus facile du parcours des options et de leurs arguments. Plusieurs cas dans le traitement des options, le cas ou ce qui est lu ne commence pas par un '-' ou '–' il est directement ajouté à la liste des fichiers. Dans le cas ou ce qui est lue correspond bien à une option enregistré, on parcours la suite de **l'iterator** le nombre de paramètre fois que possède l'option dans une autre méthode **args** qui se contente de nous renvoyé les arguments de l'option. Les cas ou le nombre d'argument n'est pas bon, c'est cette méthode **args** qui se charge de renvoyé une exception. Cependant dans le cas par exemple de l'option **-border-width** qui prend un argument de type **int** c'est lors que l'ajout dans **PaintSettings** que la vérification de si l'argument rencontré est bien un **int** ou non.

Test unitaire

- Vérifier que les options obligatoires ont bien été fournis.

- Vérifie que les options saisie sont bien connu de la librairie.

- Vérifie le bon nombre de paramètre d'une option.

Alias, Documentation, conflits

Pour gérer les alias il a fallu stocker une liste d'alias pour une option dans la classe Option.

Pour gérer la documentation on a rajouté une String correspondant à la documentation dans l la classe Option. Les documentations sont affichés grâce à la méthode usage définis dans la classe **CmdLineParser** qui fait elle même appel à la méthode de l'observer qui gère la documentation qui elle même se contente de concaténer chaque documentation si il existe des options de la **map** puis de la retourner.

Concernant les conflits un champ dans la classe Option lui est aussi consacré, c'est une Liste de String. Elle sont aussi gérer par un observer qui se contente de regarder dans la **map** si une option et ces alias porte le même nom qu'un élément dans la liste de conflits renvoie une exception si c'est le cas.

Exercice 6

- ## Le type des observers utilisé

Les **observers** dans l'exercice 6 sont pull car on notifie lorsqu'une option est enregistré, lors du process ainsi qu'à la fin du process.

- ## Avantages et inconvénients du patron observer

Les avantages du patron observer nous permet d'être informé lorsqu'il y a un changement d'état comme par exemple l'ajout d'une nouvelle option. Elle maintient notamment la cohérence des objets qui sont en relation sans pour autant créer un couplage fort entre eux.

Les inconvénients seraient qu'il faut faire pas mal de modification lorsque l'on aura besoin d'ajouter un nouvel observer.

- ## Utilité de l'interface OptionsManagerObserver

L'interface **OptionsManagerObserver** nous permet d'unifier nos **observers** en leur donnant un type commun. Le fait d'avoir **OptionsManager** comme paramètre des méthodes de l'interface nous permet d'avoir accès à l'objet et ces spécificité à l'instant ou la méthode est appelé. C'est-à-dire d'avoir un état correct de l'objet à différent moment du programme.

Par exemple à la fin de de la méthode process, il faut vérifier que les options obligatoire ont été lues pour cela nous avions du créer un observer implémentant l'interface **OptionsManagerObserver** en redéfinissant les méthode **onProcessedOption** afin de

pouvoir stocker les options lues dans une liste et la méthode **onFinishedProcess** qui regarde grâce a **l'OptionManager** si toutes les options obligatoire de l'objet **OptionManager** ont été lues.

- ## Les conflits

Les conflits sont déclarés au niveau de l'option lorsque une option est créer il est possible de saisir les conflits pour cette options qui seront stocké dans la classe Option sous forme d'une liste de String. Elles sont donc déclaré avec le nom de l'option. Ce choix permet de pouvoir saisir autant de conflits que possible pour une option puis de les traiter aisément en temps voulu en parcourant la liste des conflits pour une option.

Exercice 7

```java
public static ParameterRetrievalStrategy STANDARD = (it, option) -> {
    var args = new ArrayList<String>();

    for (var i = 0; i < option.numberParam; i++) {
        if (it.hasNext()) {
            var next :String = it.next();
            if(startWith(next, prefix: "--") || startWith(next, prefix: "-")) {
                throw new ParseException(option.name + " has no parameter", 0);
            }
            args.add(next);
        } else {
            throw new ParseException(option.name + " need " + option.numberParam +
                    " parameter(s) but got " + i, 0);
        }
    }
    return args;
};
public static ParameterRetrievalStrategy OLDSCHOOL = (it, option) -> {
    var args = new ArrayList<String>();

    for (var i = 0; i < option.numberParam; i++) {
        var next :String = it.next();
        args.add(next);
    }
    return args;
};

public static ParameterRetrievalStrategy RELAXED = (it, option) -> {
    var args = new ArrayList<String>();

    for (var i = 0; i < option.numberParam; i++) {
        var next :String = it.next();
        if (startWith(next, prefix: "-") || startWith(next, prefix: "--")) {
            return args;
        }
        args.add(next);
    }
    return args;
```

```java
public List<String> process(String[] arguments, ParameterRetrievalStrategy pStrategy) throws ParseException {
    var files = new ArrayList<String>();
    var it : Iterator<String> = Arrays.stream(arguments).iterator();

    while (it.hasNext()) {
        var next : String = it.next();
        var option : Optional<CmdLineParser.Option> = optionsManager.processOption(next);
        if (option.isPresent()) {
            var args : List<String> = pStrategy.args(it, option.get());
            option.get().action.accept(args);
        } else {
            files.add(next);
        }
    }
    optionsManager.finishProcess();
    return files;
}
@FunctionalInterface
public interface ParameterRetrievalStrategy {
    List<String> args(Iterator<String> it, CmdLineParser.Option option) throws ParseException;
}
```

Tout d'abord on créer une interface fonctionnelle contenant une méthode **args** permettant de retourner la liste des arguments d'une option.

Dans les exercices précèdent nous utilisions une méthode **args** qui se contentait de réaliser le **traîtement** de STANDARD. Cependant maintenant que l'on veut plusieurs cas différent il nous faut implémenter cette méthode **args** pour **RELAXED**, **OLDSCHOOL** et **STANDARD** qui ne change pas. Seulement pour les 2 nouvelles cas on modifie la méthode **args** afin d'accepter une option avec un arguments même si il en requiert 2 (**RELAXED**) par exemple.

- **SMARTRELAXED**

Oui il serait possible de réaliser SMARTRELAXED de la manière suivante :

```java
public static ParameterRetrievalStrategy SMARTRELAXED = (it, option, optionManager) -> {
    var args = new ArrayList<String>();

    for (;;) {
        var next : String = it.next();
        if (optionManager.byName.containsKey(option.name)) {
            break;
        }
        args.add(next);
    }
    return args;
};
```

Bilan

La librairie **CmdLineParser** contient les classes suivantes :

- **PaintSettings** → La classe contenant les options de notre librairies ainsi que les méthodes permettant de renseigné pour chaque option ces arguments si ils en ont ou bien un **boolean** qui vérifie si l'option est lu ou non.

- **CmdLineParser** → Cette classe contient plusieurs classes interne qui nous permet une accessibilité aux champs beaucoup plus facilement et ainsi réduire du code. Elle contient les classes suivante :

  - La classe Option qui contient les classe **OptionBuilder** permettant de représenter une option, ces divers paramètres.

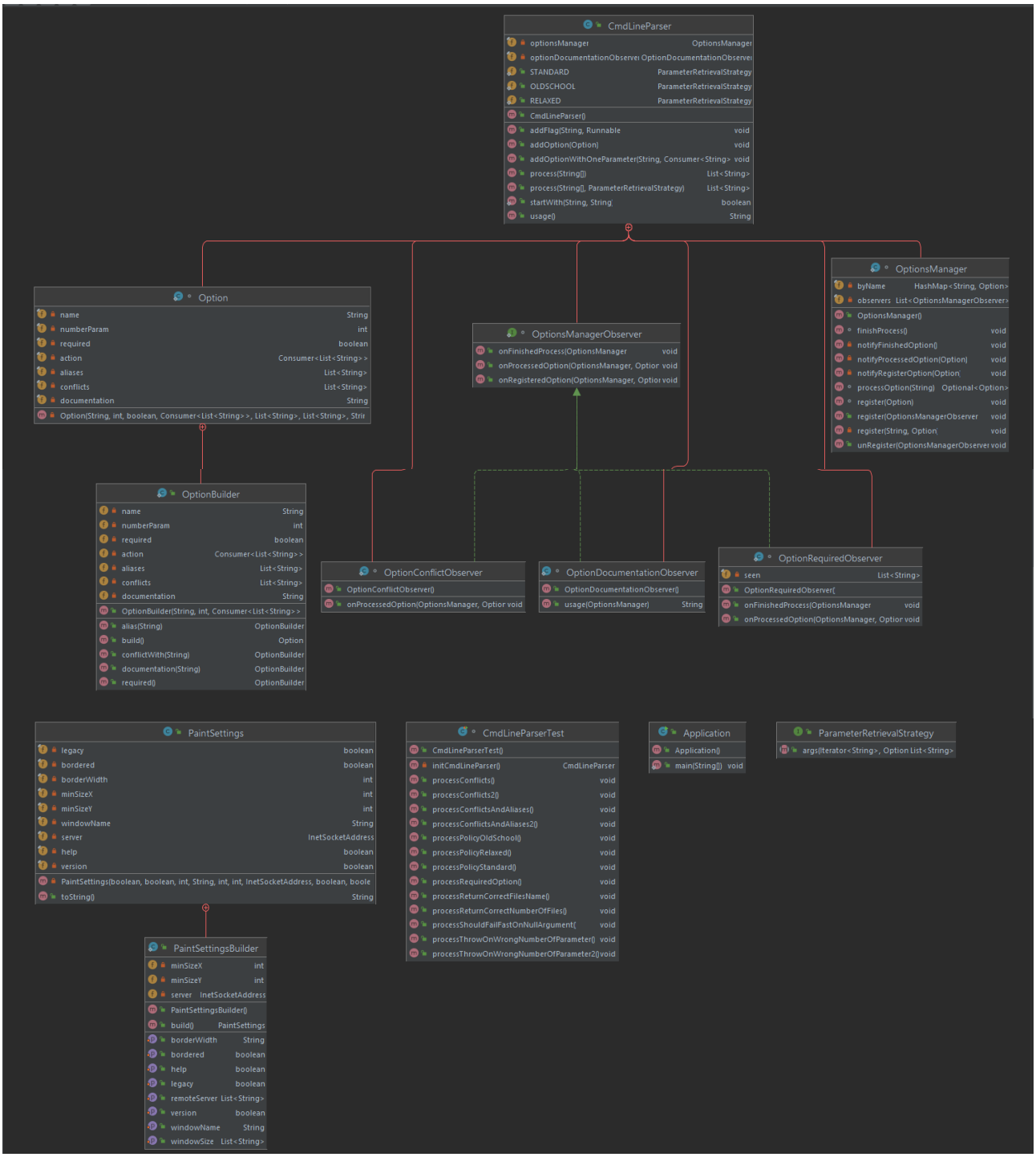
  - Une classe **OptionManager** qui gère le stockage des options avec leurs alias elle sert de classe mère pour le patron Observer. C'est cette classe qui contiendra la **map** des options enregistrée ainsi que l'enregistrement des options, le stockage des **observers** et les notifications de celle-ci.

  - Une interface **OptionsManagerObserver** qui implémente les trois méthodes de traitement d'une option c'est-à-dire son enregistrement, le processus ainsi que la fin du processus.

  Quand à la classe **CmdLineParser** en elle même elle permet de réaliser tout le processus de traitement d'une option, enregistrement, politique de traitement des options, processus des options, vérifications du respect des règle (bon nombre de paramètre pour une option, conflits ...).

- **ParameterRetrievalStrategy** → l'interface fonctionnelle afin de gérer les différentes politique de traitement des options.

- **Application** → La classe main de notre librairie.


CmdLineParser
optionsManager OptionsManager
optionDocumentationObserver OptionDocumentationObserver
STANDARD ParameterRetrievalStrategy
OLDSCHOOL ParameterRetrievalStrategy
RELAXED ParameterRetrievalStrategy
CmdLineParser()
addFlag(String, Runnable void
addOption(Option) void
addOptionWithOneParameter(String, Consumer<String> void
process(String[]) List<String>
process(String[], ParameterRetrievalStrategy) List<String>
startWith(String, String boolean
usage() String
Option
name String
numberParam int
required boolean
action Consumer<List<String>>
aliases List<String>
conflicts List<String>
documentation String
Option(String, int, boolean, Consumer<List<String>>, List<String>, List<String>, Strir
OptionsManagerObserver
onFinishedProcess(OptionsManager void
onProcessedOption(OptionsManager, Optior void
onRegisteredOption(OptionsManager, Optior void
OptionsManager
byName HashMap<String, Option>
observers List<OptionsManagerObserver>
OptionsManager()
finishProcess() void
notifyFinishedOption() void
notifyProcessedOption(Option) void
notifyRegisterOption(Option void
processOption(String) Optional<Option>
register(Option) void
register(OptionsManagerObserver void
register(String, Option void
unRegister(OptionsManagerObserver void
OptionBuilder
name String
numberParam int
required boolean
action Consumer<List<String>>
aliases List<String>
conflicts List<String>
documentation String
OptionBuilder(String, int, Consumer<List<String>>
alias(String) OptionBuilder
build() Option
conflictWith(String) OptionBuilder
documentation(String) OptionBuilder
required() OptionBuilder
OptionConflictObserver
OptionConflictObserver()
onProcessedOption(OptionsManager, Optior void
OptionDocumentationObserver
OptionDocumentationObserver()
usage(OptionsManager) String
OptionRequiredObserver
seen List<String>
OptionRequiredObserver(
onFinishedProcess(OptionsManager void
onProcessedOption(OptionsManager, Optior void
PaintSettings
legacy boolean
bordered boolean
borderWidth int
minSizeX int
minSizeY int
windowName String
server InetSocketAddress
help boolean
version boolean
PaintSettings(boolean, boolean, int, String, int, int, InetSocketAddress, boolean, boole
toString() String
PaintSettingsBuilder
minSizeX int
minSizeY int
server InetSocketAddress
PaintSettingsBuilder()
build() PaintSettings
borderWidth String
bordered boolean
help boolean
legacy boolean
remoteServer List<String>
version boolean
windowName String
windowSize List<String>
CmdLineParserTest
CmdLineParserTest()
initCmdLineParser() CmdLineParser
processConflicts() void
processConflicts2() void
processConflictsAndAliases() void
processConflictsAndAliases2() void
processPolicyOldSchool() void
processPolicyRelaxed() void
processPolicyStandard() void
processRequiredOption() void
processReturnCorrectFilesName() void
processReturnCorrectNumberOfFiles() void
processShouldFailFastOnNullArgument(void
processThrowOnWrongNumberOfParameter() void
processThrowOnWrongNumberOfParameter2() void
Application
Application()
main(String[]) void
ParameterRetrievalStrategy
args(Iterator<String>, Option List<String>