Projet GL - Partie 2

Partie 2 du projet Guava, par Samuel Payen.
Lien du dépôt : Guava

Petites modifications

Renommage

Commit du renommage.

Dans la classe *base.Optional.java* , on retrouve deux méthodes possédant le même nom, *toJavaUtil(..)*. Cependant elles ne possèdent pas la même signature, l'une ayant un argument et l'autre non. C'est donc la méthode avec un argument qui va être modifié en *toJavaUtilNullable(..)*, ce choix s'explique par l'apparition de la notation *@Nullable* dans la signature mais aussi que l'utilisation de cette méthode fonctionne dans le cas ou un argument est null. Il a également fallu modifier le nommage de la méthode pour que les tests fonctionnent toujours.

Nombre magique

Commit du nombre magique.

L'apparition du nombre magique *0xffff* en hexadécimal et l'absence d'explication explicite dans la classe *net.InetAddressses.java,* m'a conduit à rajouter une constante. En regardant le contexte de son utilisation, on en conclut que ce nombre correspond au nombre de ports maximal que possède une machine lors d'échanges d'informations entre machines. Ainsi, on peut rajouter une constante, afin faciliter la compréhension de ce nombre, comme ceci :

private static final int MAX_PORTS = 0xffff;

Ajout de documentation

Commit de l'ajout de la documentation.

Dans la classe *net.InetAddresses.java,* on remarque qu'il existe un absence non négligeable de documentation pour expliquer le fonctionnement de méthodes. Toujours dans ce contexte de réseau, il a fallu rajouter de la documentation pour plusieurs classes telles que *textToNumericFormatV4(..), textToNumericFormatV6(..)* et *parseOctet.* Ainsi, une explication du fonctionnement des méthodes ainsi que des paramètres et de la valeur de retour ont été ajoutés.

Réorganisation d'une classe

Commit de la réorganisation de la classe.

Toujours dans la classe *net.InetAddresses.java,* on remarque plusieurs imperfections dans l'agencement de la classe. Dans le but de mieux organiser cette dernière, plusieurs choix ont été faits comme :

- Déplacer les classes internes statiques *Scope* et *TeredoInfo* au début de la classe encapsulante afin de mieux les visualiser.
- Déplacer les méthodes privées après les méthodes publiques.

Moyennes modifications

Réduction cyclomatique d'une fonction

Commit de la réduction cyclomatique.

Dans la classe *net.InetAddresses.java*, la fonction *TextToNumericFormatV6* possède une complexité cyclomatique de 59, selon les métriques d'intellij. Afin de la faire baisser en dessous de 30, le refactoring par extraction de sous-méthodes m'a paru la meilleur solution.

Premièrement, une méthode a été implémentée pour vérifier que le format de l'adresse ipv6 en argument est bien conforme. La méthode *checkipv6Format* va donc vérifier que le placement des séparateurs "::" dans l'adresse et le nombre de séparateur ":" soient bien cohérents.
Enfin, un autre sous-méthode *checkNumberOfIpv6Separator* va parcourir la chaîne et vérifier que le nombre de séparateurs "::" est correct. Après ces modifications, on retrouve avec une complexité cyclomatique de la méthode principale de 19 et 19 et 28 pour les 2 autres.

Grandes modifications

Suppression d'une classe statique

Commit de suppression de la classe statique.

Dans la classe *collect.CollectCollectors.java,* j'ai supprimé la classe statique *EnumSetAccumulator*.

Modifications non faites

Suppression de code mort

Dans la partie 1, une des modifications envisagées était la suppression de la méthode *peek()* de la classe *AbstractIterator*. En effet, elle n'était appelée nulle part, cependant, en regardant de plus près son utilité, elle n'est pas inutile. Guava est une librairie utilisée par Google pour développer en Java, il faut donc faire attention à la présence de méthode considérée comme morte car elle peut être utile pour les développeurs utilisant cet API. Dans notre cas, peek() est une méthode à ne pas supprimer car son utilisation peut se révéler indispensable. Le guide d'utilisation des collections offre un exemple concret. Dans le cas où l'on veut parcourir une liste avec un *PeekingItérator* et connaître la valeur suivante sans bouger l'itérateur, la méthode peek() est indispensable car un itérateur classique ne permet pas cela.

Rassembler des classes

Les classes *Objects et MoreObjects* du package *base*, ne peuvent pas être rassemblés car la classe objects hérite de la classe *ExtraObjectsMethodsForWeb*, contrairement à l'autre classe.