Projet d'architecture

Bibliothèque de gestion de réglage applicatif

Projet réalisé par

- Belkhous Redha Nabil
- Zebouchi Mohammed

Sommaire

1. *Description du projet*

2. *Présentation de l'architecture logicielle*

3. *Patrons de conception utilisés*

4. *Présentation des fonctionnalités*

5. *jeux d'exemples et résultats*

1 Description du projet

Description du projet

Le but de l'application est de réaliser une API qui permet de sauvegarder les paramètres d'exécution pour les utilisateurs d'une application, afin que chaque partie de cette dernière puisse accéder directement au réglage qui lui convient.

Ces paramètres sont stockés dans des fichiers de telle manière que l'application récupère se dont elle a besoin uniquement.

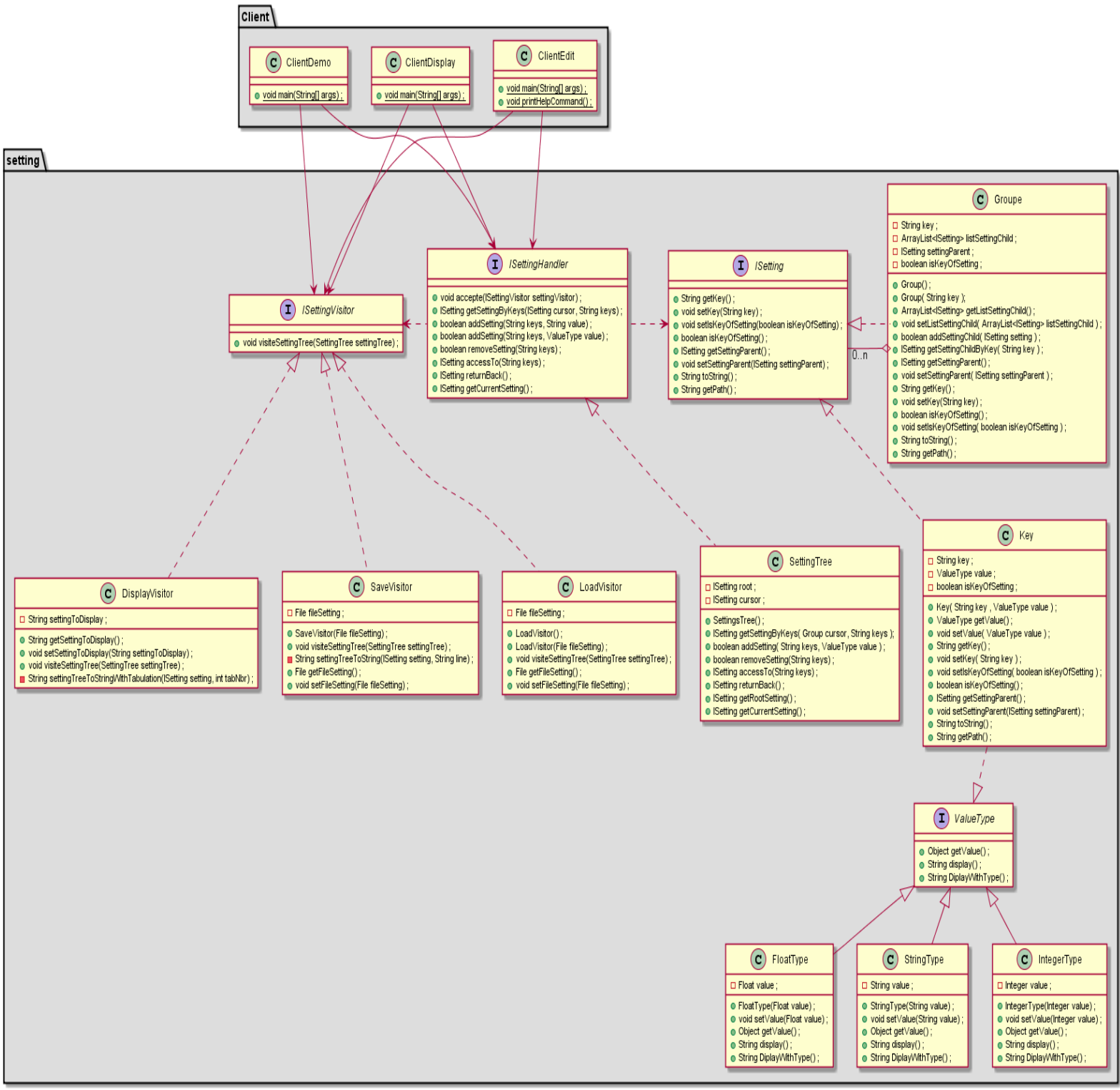
Si par exemple elle a besoin de paramétrer l'affichage d'un composant en une liste d'items selon l'environnement utilisé, on peut mettre ce réglage dans le fichier de configuration.

Ou encore pour stocker ou récupérer les identifiants et les mots de passes d'un serveur, on peut utiliser également un fichier de paramètres.

Afin de tester le bon fonctionnement de cette API, nous allons développer trois applications clientes qui permettent d'exploiter cette dernière.

2 Présentation de l'architecture logicielle

Présentation de l'architecture logicielle



On trouve dans l'architecture logicielle différentes classes et interfaces:

- La classe ClientDemo
- La classe ClientDisplay
- La classe ClientEdit
- L'interface ISetting
- L'interface ISettingVisitor
- L'interface ISettingHandler
- La classe Key
- La classe Groupe
- La classe SettingTree
- L'interface ValueType
- La classe FloatType
- La classe StringType
- La classe IntegerType
- La classe DisplayVisitor
- La classe SaveVisitor
- La Classe LoadVisitor

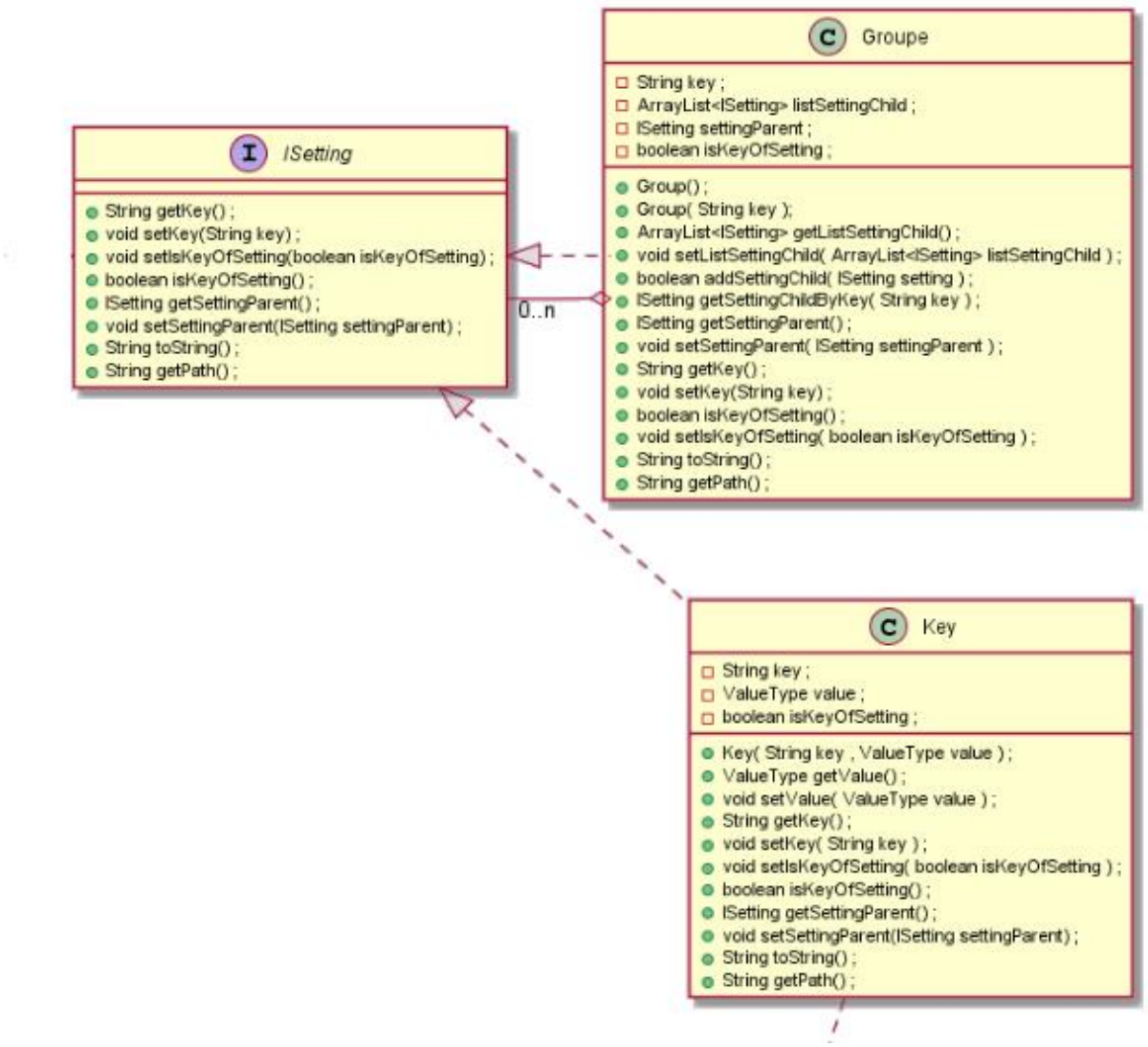
Remarque : Les détails concernant les méthodes seront donnés dans la Javadoc fournit avec le projet.

3 Patrons de conceptions utilisés

Patrons de conception utilisés

1. *Patron Composite*

Vue qu'un réglage peut être composé de plusieurs clef de groupe, qui sont concaténés entre eux avec des « **.** » et se termine avec une clef de réglage qui a une valeur. Donc nous avons opté pour le patron Composite qui permet d'avoir la relation de contenance entre les clefs. Le diagramme ci-dessus montre l'utilisation de ce dernier dans note application.



Un groupe peut contenir des sous groupes, et eux-mêmes peuvent contenir d'autres sous groupes qui se terminent par une clef ou non.

2. *Patron Visiteur*

Nous avons utilisé le patron visiteur afin de faciliter l'a manipulation des objets de différents types et pour garantir l'indépendance entre le type de stockage des réglages et les fonctionnalités globales(l'affichage, le chargement et la sauvegarde des données …etc)

4 Présentation des fonctionnalités

1. Chargement de l'ensemble des réglages à partir d'un fichier

L'application utilise le visiteur (LoadVisitor) qui prend en paramètre le fichier de réglages, un setting et un objet de stockage des réglages, dans notre cas ***settingTree***

2. Sauvegarde de l'ensemble des réglages vers un fichier

L'application utilise le visiteur (SaveVisitor) qui prend en paramètre le chemin du fichier à sauvegarder et l'objet qui contient l'ensemble des réglages. A la fin il génère un fichier qui contient l'ensemble des réglages structuré d'une manière précise.
Exemple :

```
calcul.group1.cc
calcul.group1.group2.testVal = 12
calcul.group1.group2.testVal1 = 22
calcul.group1.group2.testVal2 = zaezeaz
calcul.group1.group2.testVal3 = 12.89
calcul.group2.testVal = atzyutaz
calcul.group2.group3.text = skdjskd
calcul.group2.group3.val2 = 3763
calcul.group2.group3.group4
group2.group3.val55 = 25.33
valueIt = 22
calcul.group1.group2.testVal1 = 27
calcul.group2.group3.group4.testVal1 = 27
```

3. Obtenir une valeur à partir de sa clé de réglage

Après l'accès a un réglage précis en utilisant son chemin relatif, on peut afficher la valeur du réglage en cours grâce à la méthode « **Display** »

4. Ajouter ou modifier une valeur partir de sa clé de réglage et d'une nouvelle valeur

   Pour Ajouter une nouvelle valeur de cléf, on accès par le chemin à l'emplacement souhaité, puis on utilise la méthode « **add** » pour ajouter la clef, ou pour modifier une clef déjà existante.

5. Supprimer une valeur à partir à partir de sa clé de réglage

   La aussi, on accède à la clef qu'on souhaite supprimer et on utilise la méthode « **remove** » toute en sachant qu'on ne peut supprimer que les clefs qui viennent à partir de l'emplacement ou on est.

6. Obtenir une référence à un groupe à partir sa clé de groupe

   On utilisant la méthode « **accessTo** » de *settingHandler* qui va nous permettre d'arriver à l'emplacement souhaité et faire différent manipulation (affichage, suppression, modification ou ajout)

7. Supprimer un groupe à partir sa clé de groupe

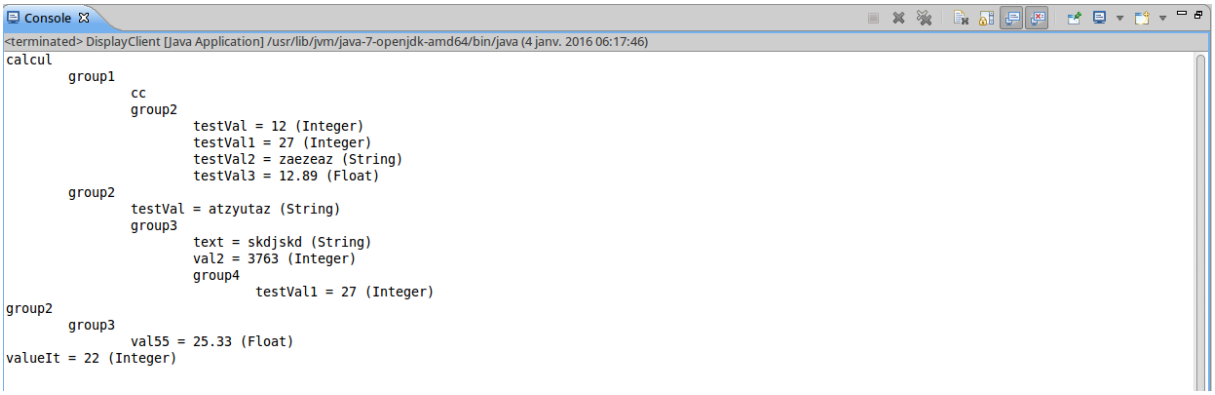
   C'est exactement le même principe que pour la suppression d'une valeur.
8. Accéder à un groupe ou une valeur de manière relative, c'est à dire à partir d'un autre groupe

   La méthode « **accessTo** » permet aussi de faire une recherche de manière relative en utilisant les parents de l'élément en court.

9. Trois clients distincts
   - Display : après avoir charger la liste de réglages, à partir d'un fichier, on affiche ces dernières de manière structurée et indentée.
   - Edit : il exploite les fonctionnalités des *visiteurs* et se *settingHandler* pour manipuler les réglages.
   - Demo : il contient les tests des fonctionnalités.

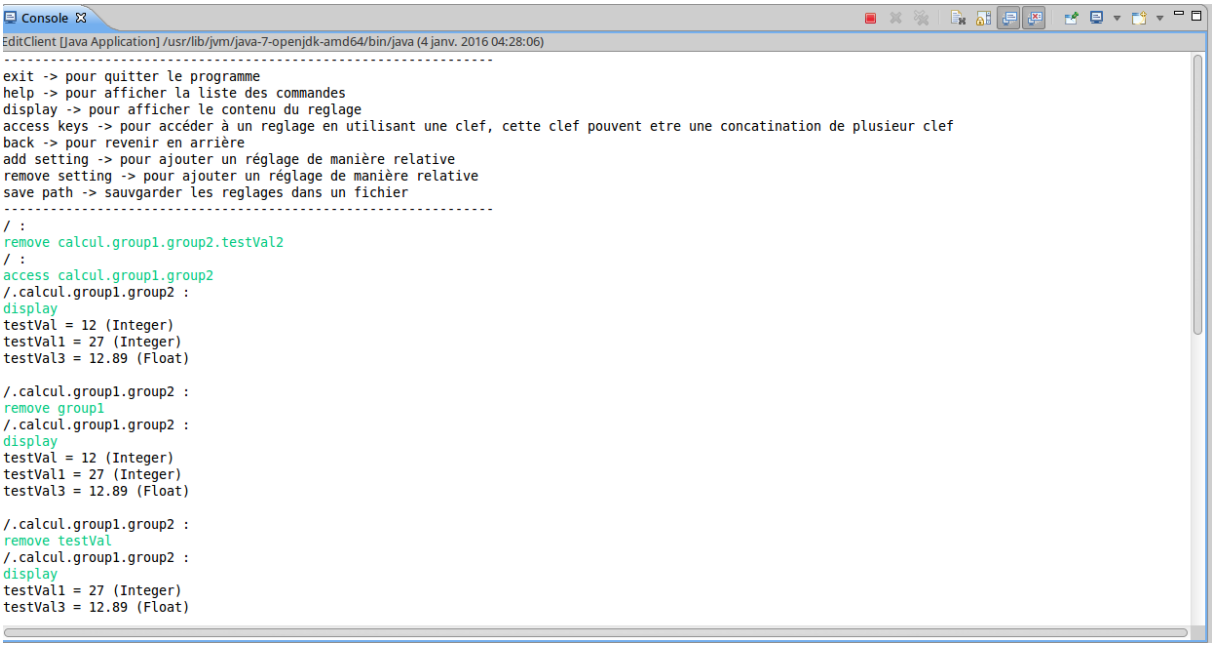
5 Jeux d'exemples et résultats

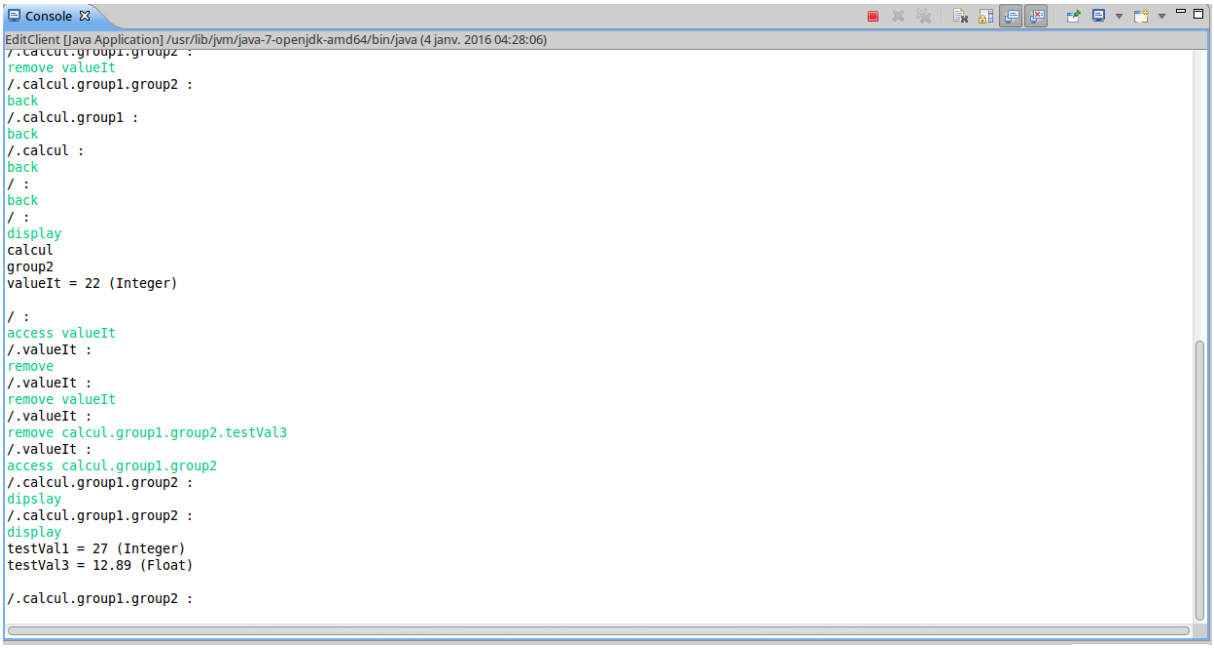
1. Display



2. Edit

Pour la manipulation des fonctionnalités, une liste de commande est à disposition de l'utilisateur, grâce à la méthode « **help** » on peut afficher ces commandes.





3. Demo

Après la manipulation des différentes fonctionnalités (ajout, suppression, accès, modification, …), on peut voir sur la capture d'écran se qui suit :