Université de Rouen

Amicale GIL

Master 1

Projet architecture logiciel

Date : 31 décembre 2015

Effectué par :
FLEURY Yoann & CROCHEMORE Valentin

Table des matières

1 Architecture de l'application

L'application est découpée en plusieurs parties :
- La partie test, avec les différents tests demandés.
- La partie façade, qui se charge de faire la liaison entre la partie test et la partie gestionnaire de paramètre.
- La partie gestionnaire de paramètre qui comporte deux sous-parties :
  - La partie d'écriture/lecture des paramètre
  - La partie gestion des clés

1.1 La partie test

En ce qui concerne la partie test, il n'y a rien de particulier à dire, chaque test a sa propre classe.

1.2 La partie façade

Comme sont nom l'indique cette partie utilise le patron façade afin de "cacher" le fonctionnement des autres parties, de faciliter l'accès au sous système et permet aussi de fournir seulement les classes nécessaire à l'exécution du client.

Le client ne sachant pas comment fonctionne le sous système, il est tout à fait possible de modifier son fonctionnement (sans changer le résultat) sans paralyser le client, par exemple si l'on veut optimiser le sous système ou changer de format d'enregistrement des paramètres. Les méthode appelé par le client redirige vers les méthodes du sous système désiré. Cela donne une certaine modularité au sous système.

Le patron façade permet de faciliter l'accès au sous système, celui-ci n'est pas compliqué d'accès, cependant en prévention d'une évolution (en taille mais aussi en complexité). Nous avons jugée que le coup de cette implémentation été minime par rapport aux gains quelle pourrait apporter.

Le patron a aussi permis de donner au client l'accès au seul classe qui lui est nécessaire pour son bon fonctionnement. Nous avons donc mis en place deux façades, chaque façade est formée par un couple interface/classe, un couple permettant seulement la "lecture seule" des réglages et couple permettant la lecture et l'écriture. Il serais tout à fait possible de rajouter un troisième couple permettant seulement l'écriture. Cela cloisonne l'accès aux classes et évite de fournir au client plus que nécessaire à son bon fonctionnement.

1.3 La partie gestionnaire de paramètre

Pour cette partie il fallait avoir une modularité assez importante afin que l'application puisse évoluer le plus facilement possible, c'est-à-dire limité le nombre de modification de l'existant au stricte minimum.

1.3.1 La partie lecture/écriture

Pour cette partie nous avons mis en place un patron proche du patron pont, dans son principe, c'est-à-dire le découplage de la partie lecture/écriture. Cela permet au deux parties d'évoluer de manière indépendantes, et d'ajouter un lecteur sans forcément ajouter d'écrivain et inversement. Cela permet aussi de faciliter le changement du type de sauvegarde des réglage (XML, XHTML ...). De plus nous avons mis en place une interface permettant aux classes plus haut niveau de manipuler des Reader et Writer sans savoir si ils s'agit de ReaderJSON ou ReaderXML par exemple. Il est donc simple de changer de format de sauvegarde.

1.3.2 La partie gestion des clés

Pour cette partie nous avons utilisé la patron composite. En effet les clés de paramètre étant découpé en deux (des clés de groupe et des clés de valeur), une clés de groupe peut contenir une autre clé de groupe ou une clé de valeur, cela correspond parfaitement au patron composite. Les clé de valeur sont donc considéré comme des feuilles et les clé de groupe comme des component. Ce patron permet au système de traiter ces deux types de clé de la même manière. Ainsi chaque clé connais la façon de s'afficher en interne ce qui simplifie l'affichage des réglages.

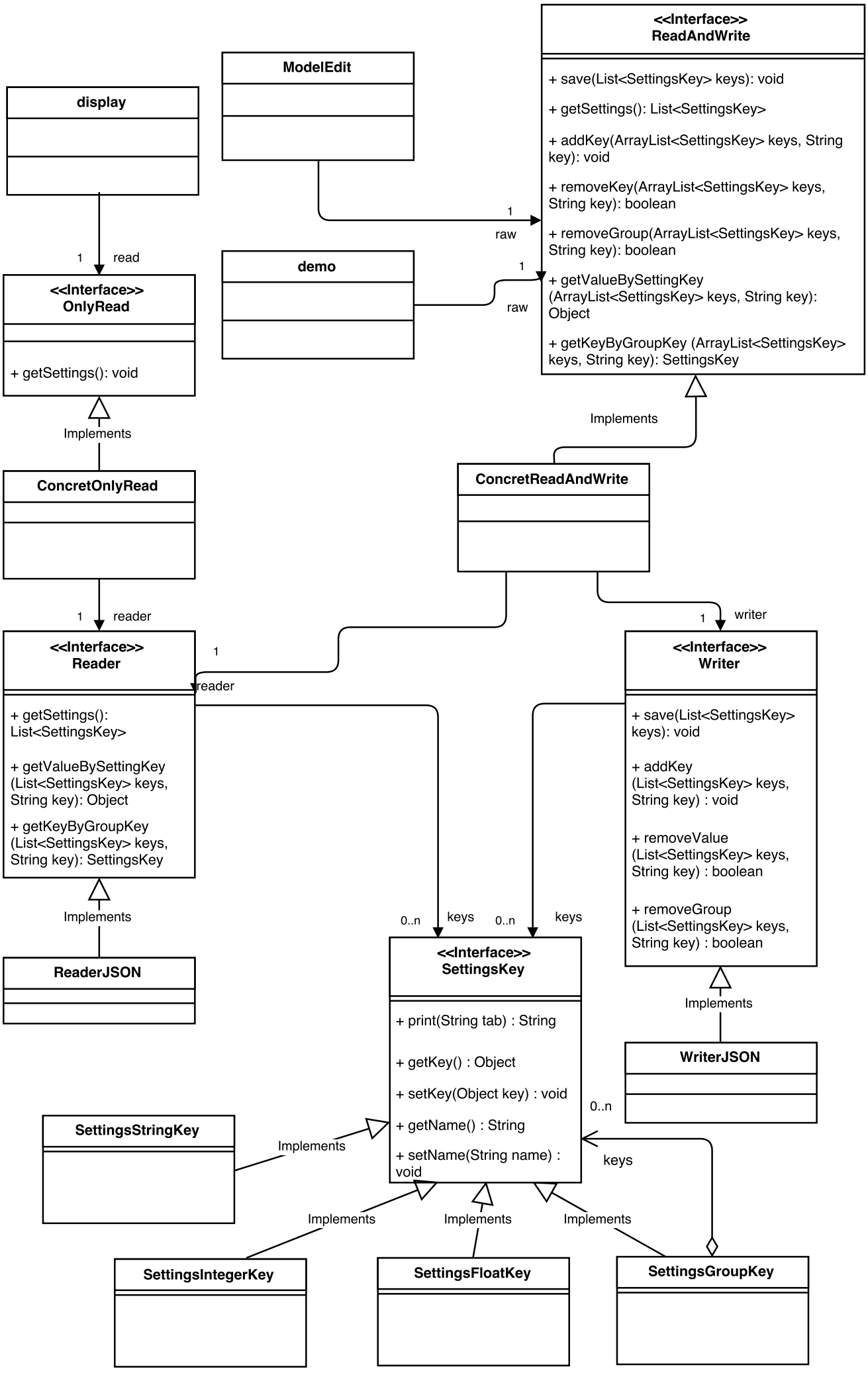
2 Fonctions implémentées

- Demander le chargement de l'ensemble des réglages à partir d'un fichier,
- Demander la sauvegarde de l'ensemble des réglages vers un fichier,
- Obtenir une valeur à partir à partir de sa clé de réglage,
- Ajouter ou modifier une valeur à partir à partir de sa clé de réglage et d'une nouvelle valeur,
- Supprimer une valeur à partir à partir de sa clé de réglage,
- Obtenir une référence à un groupe à partir sa clé de groupe,
- Supprimer un groupe à partir sa clé de groupe,
- Accéder à un groupe ou une valeur de manière relative, c'est à dire à partir d'un autre groupe.

3 Fonctions non implémentées

Toutes les fonctionnalitées ont été implémentées.

4 Schéma UML


display
ModelEdit
<<Interface>>
ReadAndWrite
+ save(List<SettingsKey> keys): void
+ getSettings(): List<SettingsKey>
+ addKey(ArrayList<SettingsKey> keys, String key): void
+ removeKey(ArrayList<SettingsKey> keys, String key): boolean
+ removeGroup(ArrayList<SettingsKey> keys, String key): boolean
+ getValueBySettingKey (ArrayList<SettingsKey> keys, String key): Object
+ getKeyByGroupKey (ArrayList<SettingsKey> keys, String key): SettingsKey
1
raw
1
read
demo
1
raw
<<Interface>>
OnlyRead
+ getSettings(): void
Implements
Implements
ConcretOnlyRead
ConcretReadAndWrite
1
reader
1
writer
<<Interface>>
Reader
1
reader
+ getSettings(): List<SettingsKey>
+ getValueBySettingKey (List<SettingsKey> keys, String key): Object
+ getKeyByGroupKey (List<SettingsKey> keys, String key): SettingsKey
<<Interface>>
Writer
+ save(List<SettingsKey> keys): void
+ addKey (List<SettingsKey> keys, String key) : void
+ removeValue (List<SettingsKey> keys, String key) : boolean
+ removeGroup (List<SettingsKey> keys, String key) : boolean
Implements
0..n
keys
0..n
keys
ReaderJSON
<<Interface>>
SettingsKey
+ print(String tab) : String
+ getKey() : Object
+ setKey(Object key) : void
+ getName() : String
+ setName(String name) : void
Implements
WriterJSON
0..n
keys
SettingsStringKey
Implements
Implements
Implements
Implements
SettingsIntegerKey
SettingsFloatKey
SettingsGroupKey