

# M1 Informatique - Ingénierie Logicielle

## TD-TP No 1 et 2

### Hiérarchies de classes, frameworks : exemple de construction et de paramétrage par spécialisation

Où l'on voit comment réaliser des classes abstraites extensibles et adaptables par spécialisation, des classes concrètes adaptant, sans les modifier, les fonctionnalités définies dans les classes abstraites. Ces entités sont au cœur des frameworks. Nous consacrerons deux séances de TP-TD au présent énoncé et une séance de TP supplémentaire pour valider la réalisation des tests. Lisez entièrement l'énoncé avant de commencer et relisez votre cours sur les schémas de réutilisation.

Cet exercice est une extension et une adaptation d'un exemple présenté dans *Smalltalk-80 : The Language and Its Implementation* de Adele Goldberg and David Robson.

## 1 Bibliothèques et Frameworks

Une bibliothèque objet est un ensemble extensible de types de données (interfaces et classes). Un *framework* est le cœur d'une application dédiée à un domaine et permettant la réalisation rapide d'applications dans ce domaine. Les premiers et les plus connus ont été les frameworks pour la réalisation d'interfaces mais il en existe aujourd'hui de très nombreux pour la réalisation d'applications distribuées ou pour le tracé de courbes ou pour la programmation par contraintes ou encore pour la réalisation de logiciels de facturation, etc. L'utilisateur d'un framework est un développeur qui va paramétrer au mieux les classes prédéfinies qui sont mises à sa disposition pour créer une nouvelle application autonome. Le paramétrage s'effectue soit par spécialisation, soit par composition soit par fonctions d'ordre supérieur (voir cours). La documentation d'un framework ou d'une bibliothèque est fondamentale, on doit indiquer aux futurs utilisateurs quels sont les endroits clés où il devra certainement créer des extensions (ici des sous-classes) et quelles méthodes il devra y définir pour paramétrer efficacement la bibliothèque ou le framework.

Les bibliothèques réutilisables et les frameworks objets sont basés sur les mêmes mécanismes suivants : liaison dynamique, polymorphisme d'inclusion et paramétrage divers. Ce premier TD/TP propose le développement d'une mini bibliothèque que l'on pourra assimiler par certains aspects à un framework.

## 2 Exemple : Cahier des charges

Un éditeur souhaite faire réaliser un programme capable de gérer "en ligne" plusieurs dictionnaires de la langue française dont : un dictionnaire complet de très grosse taille contenant toutes les définitions et un dictionnaire des mots les plus utilisés, donc plus petit, mais pour lequel les temps d'accès devront être constants et faibles. L'application doit pouvoir s'exécuter sur tous les types de machines y compris embarquées, la consommation d'espace doit donc être prise en compte.

## 3 Eléments d'Analyse

Un dictionnaire est un objet permettant de stocker des couples "clé - valeur" et de retrouver ensuite la valeur à partir de la clé. Exemple de couple : "Lavoisier - Chimiste Français, ...".

Le cahier des charges suggère la réalisation d'un programme contenant deux sortes de dictionnaires. Les premiers, destinés à contenir tous les couples "mots-définition" d'une langue contiendront beaucoup d'entrées, ils devront donc utiliser un minimum d'espace quitte à ce que l'accès aux mots

soit un peu plus lent. Les seconds, destinés aux définitions les plus utilisées, devront assurer un temps d'accès constant et faible même si cela doit coûter un peu plus d'espace mémoire.

## 4 Éléments de conception

Pour la réalisation avec un langage à objets, on prévoit de définir trois types de données (implantés par trois classes) : *OrderedDictionary*, *SortedDictionary* et *FastDictionary*. Il est possible d'y ajouter une interface, *IDictionary*

Il s'agit d'un exercice, le type *Dictionary* existe déjà en Java, il faut faire comme s'il n'existait pas.

- Pour la classe *OrderedDictionary*, les couples seront stockés de façon ordonnée par l'ordre d'insertion, la consommation d'espace disque pourra être minimale, la recherche d'éléments dans le dictionnaire sera séquentielle.
- Pour *SortedDictionary*, les couples seront stockés par ordre alphabétique sur les clés, la recherche sera également séquentielle.
- Pour *FastDictionary*, les couples seront stockés et recherchés par hachage (expliqué plus loin) sur la clé, ceci assurera un accès rapide et en temps constant aux définitions. Cela nécessitera proportionnellement plus de place mémoire, en effet une table de hachage efficace doit comporter des emplacements vides pour gérer plus efficacement les conflits. Les dictionnaires de cette catégorie ne permettront pas de retrouver les définitions dans l'ordre.

Ceci étant posé, toutes les sortes de dictionnaires précédents ont des caractéristiques communes :

- Représentation interne :

Pour l'exercice, il vous est imposé de représenter un dictionnaire par deux conteneurs de type tableau, un pour les clés, un pour les valeurs. On décide que dans tous les cas, une valeur sera rangée au même index dans le conteneur des valeurs que la clé dans le conteneur des clés. Les conteneurs seront donc des tableaux, ceci permettra de contrôler exactement leur remplissage. Ceux qui auront fini en avance pourront reprendre le TD en utilisant d'autres sortes de collections. Le but pour une instance de *OrderedDictionary* est que les conteneurs soient in fine toujours pleins pour ne pas gaspiller de place. Si l'on sait que le dictionnaire contiendra au moins  $n$  entrées, on peut initialiser la taille des conteneurs en conséquence.

Pour une instance de la classe *FastDictionary*, les contenants seront en permanence maintenus aux 3/4 pleins. L'index d'un élément sera calculé grâce à une fonction de hachage.

- Interface (c'est-à-dire, ensemble des méthodes publiques) :

Les trois types de données auront la même interface ; un utilisateur (un client) pourra écrire le même programme quel que soit le type de dictionnaire qu'il utilise.

Voici les signatures qui constituent cette interface (en d'autres termes, les instances de *OrderedDictionary*, de *SortedDictionary* et de *FastDictionary* comprendront donc les messages) :

- Object `get(Object key)`  
rend la valeur associée à `key` dans le receveur.
- Object `put(Object key, Object value)`  
entre une nouveau couple clé-valeur dans le receveur, rend le receveur.
- boolean `isEmpty()`  
rend vrai si le receveur est vide, faux sinon
- boolean `containsKey(Object key)`  
rend vrai si la clé est connue dans le receveur, faux sinon.

Exemple :

```
OrderedDictionary BD = new OrderedDictionary();
BD.put ("Lavoisier", "Chimiste francais ...");
BD.get ("Lavoisier") -->"Chimiste francais ..."
```

## 5 Réalisation dirigée

On décide de ne pas réaliser plusieurs types de données complètement indépendants mais d'essayer de partager le plus de choses possible dans une partie abstraite. Elle sert à définir tout ce qui est commun à toutes les sortes de dictionnaire afin que tout ce qui peut être écrit ne soit pas à réécrire dans la programmation des classes concrètes représentant les trois sortes de dictionnaires à implanter.

On mettra donc dans la partie abstraite une interface et une classe abstraite. Tous les points de la conception ne sont pas figés, vous devrez prendre d'autres décisions pour répondre aux questions suivantes.

### 5.1 Questions relatives au cœur de la bibliothèque

La difficulté dans la réalisation d'une classe abstraite est de déterminer quelles seront les méthodes qui peuvent y être définies (je vous les ai données) et celles qui seront spécifiques aux différentes spécialisations, dans le cas de paramétrage par spécialisation. Je vous donne également les méthodes qui devront être spécialisées.

- *indexOf(Object key)*  
rend l'index auquel est rangé le nom *key* dans le dictionnaire receveur, si *key* n'est pas dans le receveur, rend -1.
- *newIndexOf(Object key)*  
Cette méthode est appelée uniquement si *key* N'EST PAS dans le dictionnaire. Cette méthode prépare l'insertion et rend l'index auquel la nouvelle clé pourra être rangée. S'il n'y a pas assez de place dans le dictionnaire, cette méthode devra se charger d'en faire, en remplaçant les tableaux par d'autres plus grands, afin de rendre l'insertion possible. La méthode *newIndexOf* calcule ainsi l'index auquel pourra avoir lieu l'insertion, elle ne l'effectue pas elle-même.
- *size*  
rend le nombre de couples nom-valeur effectivement contenus dans le dictionnaire.

Définir l'interface `IDictionary` et la classe abstraite `AbstractDictionary` constituant la partie abstraite de la bibliothèque. Définir les méthodes de la classe `AbstractDictionary` qui peuvent l'être. Tout définir dans un package *dico*.

### 5.2 Questions sur la première spécialisation : `OrderedDictionary`

1. Définissez la classe `OrderedDictionary`. Ecrivez un constructeur sans argument et un constructeur permettant de fixer la taille initiale du dictionnaire (à condition d'être sûr du nombre de couples à insérer). Attention, pour la méthode *newIndexOf*, lorsque les conteneurs sont pleins, il faut les remplacer par des nouveaux plus grands de 1, afin de maintenir les tableaux toujours pleins.

## 6 Questions sur la seconde spécialisation : `FastDictionary`

Pour les instances de *FastDictionary*, on utilise une technique de hachage pour retrouver les index. La technique est la suivante : la méthode *hashCode* appliquée à un objet retourne un nombre (potentiellement négatif), par exemple *"toto".hashCode()* rend 6032110. Ce nombre étant potentiellement supérieur à la taille du dictionnaire, on le ramène à une valeur d'index utilisable pour le dictionnaire grâce à un modulo (opérateur : % en Java).

Ce modulo est la source des conflits de hachage. Un conflit survient à l'insertion d'un couple, lorsque l'index calculé référence un emplacement déjà occupé. En cas de conflit, on recherche à partir de l'index calculé la première place libre, en incrémentant autant de fois que nécessaire l'index, modulo la taille de la collection. Le fait que la collection soit au 3/4 pleine au maximum

assure qu'on trouvera rapidement une place libre. En résumé, le hachage permet de déterminer très rapidement la zone dans laquelle se trouve la clé recherchée, en cas de conflit, on effectue une petite recherche séquentielle locale.

1. **Taille**

Ecrivez la méthode *size* rendant le nombre d'éléments contenus dans un *fastDictionary*.

2. **Quand les tableaux doivent-ils grossir ?**

Ecrivez une méthode booléenne *mustGrow* disant si les tableaux sont au 3/4 pleins.

3. Ecrivez une méthode *grow* pour les **FastDictionary**.

4. **Mise en oeuvre**

Vous pouvez maintenant définir les méthodes permettant de spécialiser le framework à ce nouveau besoin.

Testez la classe *FastDictionary*. On doit pouvoir en créer une instance, lui ajouter des couples nom-définition et les retrouver en temps constant.

## 7 Tests

Définissez un jeu de tests couvrants pour chaque classe combinant les tests de boîte blanche et de boîte noire (cf. cours). Dotez vos classes de méthodes de test avec contrôle visuel des résultats automatisant le test du framework après chaque modification.

Vous verrez plus tard dans le cursus comment utiliser des outils d'automatisation tels que JUnit.