M1 Informatique - Ingénierie Logicielle
TD-TP 1 - séances 1 à 4

Hiérarchies de classes et Frameworks : Paramétrage par spécialisation (schéma *template method*)

Où l'on voit comment réaliser des classes abstraites et des classes concrètes adaptant, sans les modifier, les fonctionnalités (méthodes) définies dans les classes abstraites. Ces entités sont au cœur des bibliothèques de classes et des *frameworks*. Cet exercice met en oeuvre le schéma de paramétrage par spécialisation que nous édudions en cours.

Nous allons consacrer plusieurs séances de TP-TD au présent énoncé. Lisez entièrement l'énoncé avant de commencer. 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 fournit un ensemble extensible de types de données (interfaces et classes). Un *framework* est le cœur extensible d'une application dédiée à un domaine, permettant la réalisation rapide de variantes de l'applications. L'utilisateur d'un framework est un développeur qui va paramétrer les classes prédéfinies qui sont mises à sa disposition pour créer une nouvelle application autonome. Les bibliothèques extensibles et les frameworks objet sont basés sur les mêmes mécanismes de paramétrage, par spécialisation ou composition (*pattern template method*).

La documentation d'un framework ou d'une bibliothèque est importante, elle doit indiquer aux futurs utilisateurs quels sont les points de paramétrage (ou points d'extension). Ce TD/TP propose dans cet esprit la réalisation d'une classe extensible et par extension d'une d'une bibliothèque de 4 classes.

2 Exemple : Cahier des charges

Un éditeur à besoin dans ses applications de gérer "en ligne" plusieurs sortes de 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 trois 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. Ils pourront exister dans une version triée par ordre alphabétique sur les clés.

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 Eléments 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*.

Note : Il s'agit d'un exercice, le type *Dictionary* existe déjà en Java (dans un autre package), 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 sera minimale, la recherche d'une clé dans le dictionnaire sera séquentielle.
- Pour `SortedDictionary`, les couples seront stockés par ordre alphabétique sur les clés, la recherche d'une clé sera séquentielle ou dichotomique. On implantera en premier lieu la recherche séquentielle ; le premier intérêt d'avoir un dictionnaire trié est qu'il devient utilisable si on l'imprime sur sur papier.
- Pour `FastDictionary`, les couples seront stockés et recherchés par hachage (expliqué plus loin) sur la clé, ceci assurera un accès plus rapide et en temps constant aux clés et définitions. Cela nécessitera proportionellement 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 pourront pas être triés.

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 [1], 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.
  Le but pour une instance de *OrderedDictionary* est que les conteneurs soient in fine toujours pleins pour ne pas gaspiller de place.
  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.
  - IDictionary 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.
  - int size()
    rend le nombre d'éléments (donc le nombre de couples clé-valeur) contenus dans le receveur.

  Exemple :

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

5 Réalisation dirigée

On décide de ne pas programmer plusieurs types de données complètements indépendants mais d'essayer de partager le plus de choses (algorithmes, codes). La partie partagée du code définit tout ce qui est commun à toutes les classes implantant les différentes sortes de dictionnaire.

On mettra donc dans la partie partagée au moins 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.

1. Evidemment, il ne faut pas utiliser de classes telles que `Dictionary` ou `Hashtable` de *Java* pour réaliser l'exercice car elles satisfont déjà le cahier des charges. L'idée est de se placer dans la situation de ceux qui ont spécifié et codé ces classes.

5.1 Questions relatives au cœur de la bibliothèque

La difficulté dans la réalisation d'une classe abstraite (au coeur d'une bibliothèque ou d'un framework) est de déterminer quelles sont les méthodes qui peuvent y être définies (pour vous aider elles vous sont données plus haut) et comment elles sont paramétrées par d'autres. Dans le cas du paramétrage par spécialisation qui est utilisé ici, il faut donc trouver quelles sont les méthodes de paramétrage qui seront redéfinies dans les sous-classes, quelles sont leurs signatures et leurs responsabilités (ce qu'elles font). Il y a bien sûr plein de solutions possibles ; nous vous suggérons les méthodes suivantes :

- *int 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.
- *int 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, et la valeur correspondante, pourront être insérées ; elle n'effectue ces insertions.
  S'il n'y a pas assez de place pour l'insertion dans le dictionnaire concerné, cette méthode devra se charger d'en faire, en remplaçant les tableaux par d'autres plus grands, afin de rendre l'insertion possible.
- size
  rend le nombre de couples nom-valeur effectivement contenus dans le dictionnaire.

Question 1. Définir l'interface `IDictionary`.

Question 2. Définir la classe abstraite `AbstractDictionary` et déclarer ses méthodes publiques avec un corps vide.

Question 3. Mettez en oeuvre sur `AbstractDictionary` le paramétrage suggéré précédemment (ou celui de votre choix). Ceci revient à définir un ensemble de méthodes non publiques et généralement abstraites (Java), virtuelles pures (C++) ou "subclassResponsibility" (Smalltalk) sur la classe `AbstractDictionary`.

Question 4. Définissez les méthodes publiques de `AbstractDictionary`.

5.2 Questions sur la première spécialisation : OrderedDictionary

Question 5. Définissez la classe OrderedDictionary, dotée d'un unique constructeur sans paramètre. La taille initiale des conteneurs est de 0.

Notez bien que lorsque les conteneurs sont pleins, il faut les remplacer par des nouveaux plus grands de 1, afin de maintenir les tableaux toujours pleins.

Donc en résumé il faut agrandir les tableaux à chaque insertion d'un couple. Cela prend du temps à l'insertion mais cela garantit la satisfaction de la demande du client : ne pas utiliser d'espace mémoire inutilement.

Question 6. Qu'est-ce qu'une affectation polymorphique ?

Question 7. Ecrire une méthode *static main* dans une classe **UtilisationDictionnaire**, qui définit une variable `dic` de type statique `IDictionary`, qui sera affectée successivement à des instances de différents type concrets de dictionnaires ; on commencera par une affectation à une instance de `OrderedDictionary`. Dans ce main, envoyer à `dic` les messages permettant d'exécuter les 5 méthodes publiques de `IDictionary`.

Vous devez pouvoir compiler puis exécuter ce `main`.

Question 8. Mettez en place une stratégie de test plus élaborée dans le main, vérifiez que les "get" fonctionnent correctement après différents "put".

5.3 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 Java *hashCode* appliquée à un objet retourne un nombre (potentielle-

ment 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.

Question 9. Taille
Écrivez la méthode *size* rendant le nombre d'éléments contenus dans un *fastDictionary*.

Question 10. Quand les tableaux doivent-ils grossir ?
a- Écrivez une méthode booléenne *mustGrow* disant si les tableaux sont au 3/4 pleins.

b- Écrivez la méthode `grow()` de la classe **FastDictionary**.

Question 11. Mise en œuvre
Vous pouvez maintenant définir les méthodes permettant de spécialiser le framework à ce nouveau besoin.

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

5.4 Questions sur la troisième spécialisation : SortedDictionary

Les couples sont ici stockés par ordre alphabétique sur les clés. Ces dictionnaires étant triés, on peut y chercher des éléments soit séquentiellement (solution 1), soit par dichotomie (solution 2).

5.4.1 Questions dans le cadre de la solution 1

Question 12. Trouvez la solution la plus économique (du point de vue de la hiérarchie des classes et de la quantité de code écrit) pour implanter `SortedDictionary` avec une recherche séquentielle.

Étudions maintenant le problème des spécialisations en Java. Il est nécessaire que les clés dans le tableau des clés soient comparables entre elles. Le type des clés tel qu'il est déclaré dans la classe `AbstractDictionary` est trop général (`Object`). Une partie de la solution va consister à utiliser le type `Comparable` de Java. Un objet de type `Comparable` est un objet auquel on peut envoyer un message correspondant à la signature suivante : `int compareTo(Object o)` dont la description est : *Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.*

Question 13. Une première solution pour le typage est la suivante : spécialiser le tableau des clés déjà déclaré dans `AbstractDictionary` en tableau de `Comparable` dans `SortedDictionary`. Cette solution est compilable mais ne traite pas le problème : pourquoi ?

Question 14. Une seconde solution pour le typage est la suivante :

spécialiser une de vos méthode abstraites `xxx(Object key)` en `xxx(Comparable key)` sur `SortedDictionary`. Cette solution est compilable mais ne résoud pas le problème : pourquoi ?

Question 15. Implantez une solution qui résout le problème.

5.4.2 Questions dans le cadre de la solution 2

Question 16. Appliquez la solution élaborée dans le cadre de la solution 1 à la mise en œuvre de la recherche dichotomique d'éléments.