Chapitre 1 : Introduction

1.1 Structures de données

Une structure de données est une organisation logique des données qui vise à faciliter leur stockage et leur traitement.

Il est important d'étudier les structures de données car leur utilisation judicieuse est à la base de tout programme non trivial, nous retrouvons les mêmes types de structures dans tous les langages, et les algorithmes permettant leur mise en œuvre sont intéressants.

Deux préoccupations principales interviennent dans le choix d'une telle structure : la place qu'elle consomme en mémoire et la rapidité d'accès à une donnée (analyse de complexité).

Les structures de données courantes appartiennent le plus souvent aux familles suivantes : les structures linéaires (listes, piles, files, tables), les structures arborescentes (arbres binaires, arbres AVL) et les structures relationnelles (graphes orientés ou non).

1.2 Types abstraits

Pour travailler sur l'énoncé et la résolution d'un problème indépendamment d'une implantation particulière, les données sont considérées de manière abstraite : nous nous donnons une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. On parle alors de **type abstrait de données**.

1.2.1 Signature d'un type

La **signature** d'un type de données décrit la syntaxe du type (nom des opérations, type de leurs arguments) mais elle ne définit pas les propriétés des opérations du type.

Voici un exemple de signature :

```
Type Vecteur

  Opérations

       ieme : Vecteur X Entier → Element
       changer-ieme : Vecteur X Entier X Element → Vecteur
       bornesup : Vecteur → Entier
       borneinf : Vecteur → Entier
```

Remarquons que cette signature (description) du type donne sa syntaxe, mais ne suffit pas à définir celui-ci. La relative compréhension du lecteur est due au choix des noms et reste basée sur son intuition, ce qui manque pour le moins de rigueur. Pour s'en persuader, il suffit de considérer la signature ci-dessous, qui est identique, au choix des noms près :

```
Type R

  Opérations

        o : R X T → S
        p : R X T X S → R
        q : R → T
        v : R → T
```

Il apparaît alors clairement qu'une partie de la définition manque, celle qui donne une sémantique, c'est-à-dire une signification, aux noms R, S, T, o, p, q, v.

Vous noterez que le premier type qui apparaît est toujours du type défini !

1.2.2 Hiérarchie dans les types abstraits

On introduit une hiérarchie « d'utilisation » entre les types. Par exemple le type Vecteur est au-dessus des types Entier et Element dans cette hiérarchie. Cette hiérarchie est fondamentale pour structurer les définitions de type abstrait.

Elle permet d'introduire une classification importante parmi les types de données et leurs opérations :

- **Opération interne** : opération qui rend un résultat du type défini. C'est le cas pour l'opération changer-ieme du type Vecteur.
- **Observateur** : opération dont le résultat est d'un autre type prédéfini ou déjà défini comme type abstrait. C'est la cas pour les opérations ieme, bornesup et borneinf du type Vecteur.

1.2.3 Propriétés d'un type

Le problème est de donner une sémantique aux opérations de la signature.

1.2.3.1 Axiomes

Si on veut définir un type abstraitement, c'est-à-dire indépendamment de ses implantations possibles, la méthode la plus courante consiste à énoncer les propriétés des opérations sous forme d'**axiomes**.

Par exemple,

```
borneinf(v) ≤ i ≤ bornesup(v) => ieme(changer-ieme(v,i,e),i) = e
      où v, i et e sont des variables respectivement
       du type Vecteur, Entier et Element.
```

Cet axiome exprime que, dans la mesure où i est compris entre les bornes d'un vecteur v, quand on construit un nouveau vecteur en donnant au ième élément la valeur e, et que l'on accède ensuite au ième élément de ce nouveau vecteur, on obtient e. Cette propriété est satisfaite quelles que soient les valeurs, correctement typées, données aux variables.

Un autre axiome serait:

```
borneinf(v) ≤ i ≤ bornesup(v) & borneinf(v) ≤ j ≤ bornesup(v) & i ≠ j
=> ieme(changer-ieme(v,i,e),j) = ieme(v, j)
```

Cet axiome combiné avec le précédent exprime que seul le ième élément a changé dans le nouveau vecteur. Ces deux propriétés doivent être satisfaites par toute implantation du type abstrait Vecteur.

La définition d'un type abstrait de données est donc composée d'une signature et d'un ensemble d'axiomes. Les axiomes sont accompagnés de la déclaration d'un certain nombre de variables. Ce type de définition s'appelle une **spécification algébrique** ou **axiomatique** d'un type abstrait. Pour abréger on parle souvent de **types abstraits algébriques.**

1.2.3.2 Consistance

N'y-a-t-il pas d'axiomes contradictoires ?

Un exemple d'inconsistance serait, dans le cas de vecteurs d'entiers, de trouver une expression v de type Vecteur et une expression entière i telles que l'on puisse démontrer, en utilisant les axiomes, les deux propriétés : **ieme(v, i) = 0** et **ieme(v, i) = 1** .

1.2.3.3 Complétude

Y-a-t-il un nombre suffisant d'axiomes ?

La notion de complétude est plus délicate et demande d'être examinée avec soin quand on travaille sur des types abstraits. En mathématiques, une théorie T, et par suite le système d'axiomes qui la définit, est dite **complète** si elle est consistante et si pour toute formule P sans variable, on sait démontrer soit P soit non P.

Cette notion est trop forte pour les types abstraits algébriques. *Elle entraîne que toute égalité de deux expressions sans variable doit être soit vraie, soit fausse. Or souvent :* on veut et on doit laisser une latitude aux développeurs.

Le critère utilisé pour les types abstraits algébriques est la complétude suffisante: **les axiomes doivent permettre de déduire une valeur d'un type prédéfini pour toute application d'un observateur à un objet d'un type défini.**

Comme on obtient les objets par les opérations internes, **il faut écrire des axiomes qui définissent le résultat de la composition des observateurs avec toutes les opérations**

internes, et ce sur le domaine de définition des observateurs. De tels axiomes ont été donnés pour **ieme** et **changer-ieme**, dans l'exemple des vecteurs.

Le domaine de définition d'une opération partielle est défini par une **précondition**.

Par ailleurs, on n'a pas la possibilité avec la signature actuelle de construire un nouveau Vecteur : la seule opération interne est **changer-ieme** qui prend en argument un vecteur. Il faut donc ajouter une opération interne qui correspond au contenu d'un vecteur dont les éléments sont initialisés à 0 et dont on fixe les bornes :

```
nouveau : Entier X Entier → Vecteur
      avec les axiomes :
              0 <= i <=j => borneinf(nouveau(i, j)) = i
              0 <= i <= j => bornesup(nouveau(i, j)) = j
              0 <= i <= j  & i ≤ k ≤ j => ieme(nouveau(i,j),k)=0
```

Il manque les définitions des observateurs **bornesup** et **borneinf** sur le résultat de l'opération interne **changer-ieme**:

```
borneinf(v) ≤ i ≤ bornesup(v)
            => borneinf(changer-ieme(v, i, e)) = borneinf(v)

borneinf(v) ≤ i ≤ bornesup(v)
            => bornesup(changer-ieme(v, i, e)) = bornesup(v)
```

1.3 Exemple complet

La définition finale du type Vecteur est donnée ci-après. Elle est suffisamment complète. Tout vecteur est le résultat d'une opération **nouveau** et d'une suite d'opérations **changer-ieme**. Les axiomes permettent de déduire le résultat de **ieme**, **bornesup** et **borneinf** dans tous les cas.

En conclusion un critère pour savoir si on a écrit suffisamment d'axiomes est: **peut-on déduire de ces axiomes le résultat de chaque observateur sur son domaine de définition ?**

```
Type Vecteur

      Opérations

            nouveau : Entier X Entier → Vecteur
            changer-ieme : Vecteur X Entier X Element → Vecteur
            ieme : Vecteur X Entier → Element
            borneinf : Vecteur → Entier
            bornesup : Vecteur → Entier

      Préconditions

             nouveau(i,j) défini_ssi 0<=i<=j
             changer-ieme(v,i,e) défini_ssi borneinf(v) ≤ i ≤ bornesup(v)
             ieme (v,i) défini_ssi borneinf(v) ≤ i ≤ bornesup(v)
```

```
Axiomes

     ieme(nouveau(i,j),k)=0
     ieme(changer-ieme(v, i, e), i) = e
     i ≠ j => ieme(changer-ieme(v, i, e), j) = ieme(v, j)

     borneinf(nouveau(i, j)) = i
     borneinf(changer-ieme(v, i, e)) = borneinf(v)

     bornesup(nouveau(i, j)) = j
     bornesup(changer-ieme(v, i, e)) = bornesup(v)
```

Remarque : certaines conditions sur les axiomes ont été traduites en préconditions.

☺ **Exercice 1.A :** Ecrire la spécification algébrique du type Ensemble [T].

1.4 Du type abstrait à la classe

Les classes concernent les langages à objets. A chaque type nous pouvons associer une classe. En programmation types et classes deviennent synonymes (classe = type). Les opérations du type se traduisent (se réalisent ou encore s'implantent) par les méthodes de la classe.

De manière informelle, une classe est un élément logiciel qui décrit un type de données et son implantation totale ou partielle.

La traduction des opérations s'opère comme suit :

- Le premier argument d'une opération, **qui est du type défini**, devient le receveur, c'est-à-dire l'objet sur lequel on applique la méthode. Les autres arguments sont les paramètres de la méthode.
- Les méthodes qui renvoient un résultat du type modifieront l'objet courant.
- On a besoin également d'opérations de construction d'objets : les constructeurs.

1.5 Rappels sur la généricité en java

La généricité permet d'écrire des collections dont tous les éléments ont le même type, sans avoir à répéter plusieurs fois le même code, en vérifiant le typage dès la compilation et en évitant les transtypages (casts) à l'exécution.

Un type générique est défini comme une classe ou une interface paramétrée par une section de paramètres de la forme <T1, T2,…, Tn>. Ces paramètres représentent des types inconnus au moment de la compilation du type générique. On place la liste des paramètres à la suite du nom de la classe ou de l'interface : **Boite<E>**, **Association<K,V>**. Ils peuvent être utilisés dans le code du type générique, excepté pour créer des objets, pour définir un super-type d'un autre type, ou pour définir un attribut statique.

☺ **Exercice 1.B** : Écrire une classe générique Triplet permettant de manipuler des triplets d'objets d'un même type. On la dotera d'un constructeur à trois arguments (les objets constituant le triplet), et de trois méthodes d'accès getPremier, getSecond et getTroisieme, permettant d'obtenir la référence de l'un des éléments du triplet. Ecrire également une classe de test utilisant cette classe générique pour instancier quelques objets et exploiter les méthodes existantes.

☺ **Exercice 1.C** : Traduire le type Ensemble [T] sous la forme d'une classe Java portant le même nom. Vous écrirez une classe de tests simple qui permettra d'instancier cette classe.

☺ **Exercice 1.D** : Même travail que l'exercice 1.B, mais cette fois-ci en écrivant une classe générique TripletH permettant de manipuler des triplets d'objets pouvant être chacun d'un type différent.

☺ **Exercice 1.E** : Repérer les erreurs commises dans les instructions suivantes.

```
public class C<T> {
          T x ; T[] t1 ; T[] t2 ;
          static T inf ; static int compte ;
          void f () { x = new T () ;
                      t2 = t1 ;
                      t2 = new T [5] ;
                    }
}
```

1.6 Test des implantations de types abstraits

Avant de se lancer dans l'étude des implantations possibles d'un type abstrait, il faut réfléchir aux tests que l'on se propose d'écrire pour tester ces futures implantations.

Les tests peuvent être construits selon la technique dite **boîte noire** ou **boîte blanche** : les premiers sont indépendants de l'implantation et s'intéressent aux comportements des méthodes ; les seconds vérifient chaque implantation.

Nous nous intéressons ici aux techniques s'appliquant aux tests boîte noire s'appuyant sur la spécification algébrique (*parce que nous ne pouvons pas tout faire*).

1.6.1 Problèmes posés

Il est important d'optimiser la phase de tests (fastidieuse, comme chacun le sait ...) et de faire en sorte que les mêmes tests soient faits pour les différentes implantations, sans être obligé de tout réécrire pour chacune d'entre elles,

Il est également crucial d'obtenir une couverture des tests suffisante pour qu'on ait l'intime conviction qu'il ne reste pas de bugs (à défaut de pouvoir le prouver complètement ...),

La lecture des résultats des tests n'est pas à négliger : les lignes générées sur la sortie standard doivent être claires sur la nature du bug et synthétiques pour faciliter la lecture.

1.6.2 Méthodologie

Ecrire des tests au petit bonheur la chance, au gré de son humeur et de son envie, n'offre sûrement pas une couverture suffisante et ne produit que l'illusion que les classes ont été testées. Un peu de rigueur s'impose ...

Dans un premier temps, on s'appuie sur le fait que le comportement de toutes les implantations est fixé par une spécification commune. Ainsi, il faut chercher à mutualiser l'effort de conception d'une classe de test, de sorte qu'on puisse tester de façon similaire plusieurs implantations, sans avoir à tout refaire.

Dans le cas d'implantation multiple, nous pouvons créer une hiérarchie de classes de tests parallèle à la hiérarchie des classes correspondant aux différentes implantations. Par exemple, si les classes **EnsembleImpl1** et **EnsembleImpl2** implantent l'interface **Ensemble** correspondant au type **Ensemble**, on crée deux classes de tests, appelées **TestEnsembleImpl1** et **TestEnsembleImpl2**. Si on est amené à ajouter une nouvelle implantation de **Ensemble**, on ajoutera une nouvelle classe de test dans la hiérarchie.

Dans ce cas, les classes de tests hériteront alors toutes de la classe **TestEnsemble**, qui met en commun la structure générale des tests. Cela permet de minimiser l'effort de développement d'une nouvelle classe de test pour une nouvelle implantation.

Pour structurer les tests, il est préférable que la classe **TestEnsemble** appelle une méthode spécifique par méthode à tester : par exemple, la méthode **testCardinal** gère uniquement les tests associés à l'opération **cardinal**. Cela évite d'écrire une (trop) longue méthode main.

La structure de la classe de test **TestEnsemble** peut être la suivante :

```
abstract class TestEnsemble {

      public static void main ( String [ ] a) {

            TestEnsemble t = new TestEnsemble() ;
            t.testCardinal() ;
            t.testAppartient();

      }
}
```

Pour réaliser les tests proprement dits d'une fonction donnée, nous nous appuyons sur la spécification algébrique qui décrit de façon précise et complète le comportement de la fonction. Chaque fonction de test est écrite sur le même modèle, en testant les axiomes les uns après les autres et surtout indépendamment les uns des autres. Ainsi, la structure de la méthode **testCardinal** ressemble à :

```
public void testCardinal() {
      // cardinal(vide()) = 0
```

```
      // cardinal(ajouter(ens, el)) = cardinal(ens)+1
      …
} // testCardinal()
```

On pourrait imaginer de décomposer encore et de faire finalement une fonction de test par axiome ; peu importe. L'indépendance entre les tests est, elle, primordiale pour obtenir une couverture suffisante et ne pas se placer dans un cas particulier de test.

Pour garantir l'indépendance des tests, les objets construits pour vérifier un axiome ne doivent en aucun cas être réutilisés pour tester un autre axiome.

De façon générale, la séquence de test de l'axiome gauche = droit s'écrit sous la forme :

```
      Construction de gauche
      Construction de droit
      Si gauche est différent de droit, déclencher une exception
```

A noter que l'ordre de construction des termes gauche et droit peut être quelconque puisqu'il s'agit d'une équation. On verra à l'usage que dans certains cas, un ordre de construction s'impose de lui-même.
La construction de cette séquence pose deux problèmes essentiels, liés au fait que dans un axiome, gauche et droit sont des termes fonctionnels : comment construire un objet à partir d'une définition fonctionnelle ? Comment comparer les deux objets obtenus ?

Le test du premier axiome de la fonction cardinal ne pose pas de problème, car il n'existe qu'un seul objet ensemble vide. La séquence de tests ci-dessous fait appel à une méthode **initialisationVide** ou **setUpVide**, chargée de créer un ensemble vide dans la variable ens. Cette fonction peut être abstraite, implantée de façon spécifique dans chaque sous-classe de test (ce qui implique alors que la classe **TestEnsemble** est abstraite). Ensuite l'instruction **assert** permet de tester explicitement l'axiome : si la condition n'est pas vérifiée, une exception est déclenchée (avec le message spécifié).

```
public void testCardinal ( ) {

      // cardinal(vide()) = 0
      Ensemble<Integer> ens = initialisationVide () ;
      int gauche = ens.cardinal() ;
      assert(gauche == 0): "Bug cardinal(vide()) = 0" ;
      …
}
```

Rappel : **assert Expression1 : Expression2**. Lorsque l'expression booléenne Expression1 est fausse, le système affiche le message d'erreur Expression2. L'option -ea permet de prendre en compte les assertions lors de l'exécution d'un programme (java –ea).

Le second axiome **cardinal(ajouter(ens, el)) = cardinal(ens)+1** fait intervenir une variable ens supposée quelconque. Il est impossible de tester exhaustivement tous les objets possibles. Une solution couramment appliquée consiste à identifier des classes d'équivalence de ces objets, ayant a priori un comportement commun.

On se contente alors de tests sur des objets pris au hasard dans chaque classe d'équivalence. Se limiter à des classes d'équivalence est bien sûr restrictif ; plus on fait de classes d'équivalence, plus on fait de tests, plus on se rapproche du zéro-bug.

Dans le cas des ensembles, on peut imaginer **3 classes d'équivalence** : la classe des ensembles vides (aucun élément), la classe des ensembles à un élément et la classe des ensembles, de plus d'un élément, car on imagine que dans chacune de ces trois classes, les ensembles se comportent de la même façon.

Donc, pour tester le second axiome de la fonction cardinal, on écrit trois séquences différentes, correspondants aux trois classes d'équivalence.

```
public void testCardinal () {
      …
      // cardinal(ajouter(ens, e1)) = cardinal (ens) + 1
            // cas où ens est vide
            // cas où ens est réduit à un élément
            // cas où ens contient plus d'un élément
      … }
```

Ce découpage va se retrouver de façon systématique dans chaque fonction de test : pour le cas où l'ensemble est vide, on utilise la méthode **initialisationVide** qui crée l'objet à tester et pour les deux autres classes d'équivalence, on utilise de la même façon deux autres méthodes **initialisationUnElement** et **initialisationPlusieursElements**.

```
public void testCardinal ( ) {
      …
      // cardinal(ajouter(ens, e1)) = cardinal (ens) + 1

      // cas où ens est vide
      Ensemble<Integer> ens = initialisationVide() ;
      int gauche = ens.cardinal()  + 1 ;
      Integer el ;
      while (ens.appartient(el)) {  // pour garantir la précondition
      el= new Integer ((int)(Math.random( )*100)) ;}

      int droit = ens.ajouter(el).cardinal() ;
      assert (gauche == droit) :
             "bug (ajouter(ens, e1)) = cardinal (ens) + 1 avec ens vide"

      // cas où ens est réduit à un élément
      …
      // cas où ens contient plus d'un élément
      …
}
```

Remarque : dans le cas du second axiome, un ordre est imposé dans la construction des deux membres. **cardinal(ens)** doit être évalué avant l'ajout de l'élément el.

On peut noter que cette technique produit un programme de test très long ; néanmoins, l'objectif n'est pas de faire court mais de tester le plus largement possible avec la meilleure fiabilité possible.

☺ **Exercice 1.F :** Construction de la spécification algébrique du type Tarif et écriture d'un programme de test à partir de la spécification.

Une application commerciale doit gérer un tarif, c'est à dire une collection de produits étiquetés par un prix, en utilisant différentes opérations permettant de :

- s'assurer de l'existence d'un produit dans le tarif,
- connaître le prix d'un produit existant dans le tarif,
- ajouter un nouveau produit, avec son prix,
- modifier le prix d'un produit, supprimer un produit,
- fusionner deux tarifs, en conservant le prix le plus faible si le même produit fait partie des deux tarifs,
- identifier les produits répertoriés dans deux tarifs en même temps.

Question 1.F.1 : écrire la spécification algébrique du type abstrait algébrique Tarif (opérations, préconditions et axiomes).

Question 1.F.2 : écrire une classe de test pour l'implantation de Tarif, en s'appuyant sur la spécification algébrique (traduction des axiomes).

Question 1.F.3 (pour ceux qui avancent vite) : proposer une implantation de la classe Tarif.