Assertions et tests : Une introduction par la mise en oeuvre d'ensembles génériques de style OCL (`OclCollections.Set`) en Java 5.0 avec JUnit 4.0

Hiver 2010

Table des matières

1 Introduction

Dans ce document, nous allons voir de quelle façon les «assertions», qui sont essentiellement des *affirmations de propriétés logiques*, donc des *propositions*, peuvent être utilisées pour *tester des programmes*. Plus spécifiquement, cette présentation se fera en examinant une partie de la mise en oeuvre, en Java 5.0, d'une bibliothèque de collections génériques semblable aux collections OCL. Quant à l'aspect des tests, il sera abordé en présentant le cadre de tests JUnit 4.0.

2 Évaluation dynamique des assertions

2.1 L'instruction `assert`

Le langage Java possède une instruction `assert`. Cette instruction permet d'évaluer de façon dynamique — c'est-à-dire à l'exécution — une expression booléenne (une proposition) et de signaler une exception si le résultat obtenu est `false` ; par contre, l'instruction n'a aucun effet si l'expression est `true`. Plus précisément, ces assertions sont évaluées uniquement si l'option «`-ea`» — ou «`-enableassertions`» — est utilisée lors de l'exécution du programme, donc lors de l'appel de la machine virtuelle Java.

L'instruction `assert` possède deux formes :

- `assert` $expr_1$`;`
- `assert` $expr_1$ `:` $expr_2$`;`

Si l'évaluation dynamique des assertions est activée (option `-ea`), alors l'effet de chacune de ces instructions est le suivant :

- `assert` $expr_1$`;` : L'expression booléenne $expr_1$ est évaluée. Si la valeur résultante est `true`, alors on passe à l'instruction suivante. Si la valeur résultante est `false`, alors l'exception `java.lang.AssertionError` est signalée.
- `assert` $expr_1$ `:` $expr_2$`;` : Même processus d'évaluation que la forme précédente, à la différence que la valeur de $expr_2$, qui doit être une chaîne de caractères, est transmise comme argument au constructeur de l'exception, et donc apparaîtra dans le message d'erreur de l'exception.

2.2 Un exemple de programme Java avec des assertions

Le Fichier Java 1 présente un petit exemple de programme Java contenant diverses instructions `assert`. La figure 1 présente les résultats pour différentes exécutions du programme.

2.3 Les différents rôles des assertions

Les assertions peuvent être utilisées pour remplir différents rôles — nous examinerons plusieurs de ces rôles dans la suite du cours :

- Pour spécifier des pré-conditions ;
- Pour spécifier des post-conditions ;
- Pour spécifier des invariants ;
- Pour rendre explicites des hypothèses ou suppositions implicites ;
- Pour aider au débogage d'un programme ;

Fichier Java 1 Programme Java illustrant les assertions : `ExemplesAssertions.java`.

```
class ExemplesAssertions {

  public static void main( String[] args ) {

    if (args.length == 0) {
      System.out.println( "Usage:" );
      System.out.println( "  java ExemplesAssertions n" );
      System.exit(1);
    }

    int n = Integer.parseInt( args[0] );

    assert n > 0;

    assert n < 10;

    assert n % 2 == 0 : "n = " + n + " pas divisible par 2";

    System.out.println( " n = " + n );

  }
}
```

- Pour vérifier des conditions supposément impossibles;
- Pour tester un programme.

Dans ce qui suit, nous allons examiner ce dernier aspect, à savoir, comment tester un programme en utilisant des assertions. Dans un premier temps, quelques brèves explications sur les tests.

3 Rôle des tests et tests unitaires

Note : Cette section est un *résumé* et *adaptation* d'un document sur les tests présentés dans le cours INF3135 «Construction et maintenance de logiciels»[1] .

- *Test Your Software, or Your Users Will.*
- *Test Early. Test Often. Test Automatically.*
- *Coding Ain't Done 'Til all the Tests Run.*

"The Pragmatic Programmer" [HT00]

A. Hunt & D. Thomas

3.1 Les différents types de tests (que faut-il tester?)

Les différents niveaux de tests [RK03]

- Tests unitaires :
  - Un test unitaire vérifie le bon fonctionnement *d'un module*, d'un *composant* indépendant — dans le contexte Java, d'une classe ou petit groupe de classes.

[1] `http://www.info2.uqam.ca/~tremblay/INF3135/Materiel/`

```
% javac ExemplesAssertions.java

% java ExemplesAssertions
Usage:
  java ExemplesAssertions n

% java ExemplesAssertions 0
 n = 0

% java ExemplesAssertions 2
 n = 2

% java ExemplesAssertions 3
 n = 3

% java ExemplesAssertions 99
 n = 99

% java -ea ExemplesAssertions
Usage:
  java ExemplesAssertions n

% java -ea ExemplesAssertions 0
Exception in thread "main" java.lang.AssertionError
    at ExemplesAssertions.main(ExemplesAssertions.java:13)

% java -ea ExemplesAssertions 2
 n = 2

% java -ea ExemplesAssertions 3
Exception in thread "main" java.lang.AssertionError: n = 3 pas divisible par 2
    at ExemplesAssertions.main(ExemplesAssertions.java:16)

% java -ea ExemplesAssertions 99
Exception in thread "main" java.lang.AssertionError
    at ExemplesAssertions.main(ExemplesAssertions.java:14)
```

Figure 1: Résultats de la compilation et de diverses exécutions du programme `ExemplesAssertions.java`.

  - Les tests unitaires forment la *fondation* sur laquelle repose l'ensemble des activités de test : inutile de tester le tout si les parties n'ont pas été testées à fond.

- Tests d'intégration :
  - Les tests d'intégration vérifient que les principaux *sous-systèmes* fonctionnent correctement, c'est-à-dire que les différents modules qui composent un sous-système donné sont correctement *intégrés* ensemble.
  - Les tests d'intégration peuvent être vus comme une forme de tests unitaires, l'unité étant alors un *groupe (cohésif) de modules* plutôt qu'un unique module.

- Tests de système :
  - Les tests de système vérifient le fonctionnement du système dans son ensemble, en termes des fonctionnalités attendues au plus haut niveau.

- Tests d'acceptation :
  - Les tests d'acceptation sont des tests finaux effectués lorsque le système est prêt à être déployé, juste avant qu'il soit livré et installé.
  - L'objectif est de vérifier que le logiciel est prêt à être utilisé par les véritables usagers, pour effectuer les tâches et fonctions pour lesquelles le système a été développé.

Tests unitaires

- Les *cas de test* pour les tests unitaires d'un module sont définis relativement au *contrat* ce module :
  - Quelles sont les pré-conditions qui régissent l'utilisation de ce module?
  - Quelles sont les post-conditions qu'assurent l'exécution de ce module?
  - Quelles sont les exceptions signalées par ce module en présence de cas spéciaux ou anormaux?

- Les tests unitaires pour un module sont intimement liés au code source de ce module. En fait, de nos jours, on considère que le *code source* pour un module consiste *autant* dans le code réalisant ce module que dans le code effectuant les *tests unitaires* de ce même module, donc :

  > *Code d'un module = code réalisant les fonctionalités du module + code réalisant les tests*

  En d'autres mots, les tests forment un «livrable» aussi important que le code lui-même (et il faut donc s'assurer de bien intégrer les deux).

- Autre avantage des tests unitaires intégrés au code : les cas de test fournissent des exemples illustrant la façon d'utiliser le module.

3.2 Les principales activités liées aux tests (comment faut-il tester?)

3.2.1 La spécification des tests

Une des façons de spécifier des jeux d'essai [RK03] est d'utiliser des données *typiques* et réelles, comprenant aussi des cas erronés et anormaux. Les méthodes et techniques pour identifier les tests qui doivent être développés seront abordés dans d'autres cours, par exemple, INF3135 «Construction et maintenance de logiciels» ou INF6160 «Qualité : Processus et produit». Soulignons simplement qu'un ensemble de jeux d'essai devrait couvrir les différents cas suivants— voir la citation de B. Meyer à ce sujet, section C (p. 42) :

- Cas normaux — valeurs clairement valides.
- Cas anormaux — valeurs clairement invalides.
- Cas frontières — valeurs aux limites des cas valides/invalides.

Par exemple, si on veut définir une fonction qui vérifie qu'un nombre est bien compris entre 0 et 100 inclusivement, il serait bon d'avoir des cas de test couvrant les différentes possibilités suivantes :

- Cas normaux : 10, 90 ;
- Cas anormaux : -10, 110 ;
- Cas limites : -1, 0, 1, 99, 100, 101.

3.2.2 L'exécution des tests

Les différents cas de test, une fois définis, doivent pouvoir être exécutés avec le logiciel qu'on veut tester. De nos jours, on considère important que les tests puissent être exécutés, *et vérifiés*, de façon automatique— par *vérifier*, on veut dire ici *s'assurer que le résultat produit est bien celui attendu.*

Pourquoi il est important que les tests soient exécutés de façon automatique

- Il est important de s'assurer, lorsqu'une ou plusieurs modifications ont été effectuéesà un module ou au programme, que rien n'a été brisé, que ce qui fonctionnait correctement auparavant fonctionne toujours après que les modifications ont été faites. On parle alors de *tests de régression.*[2]
- Il faut pouvoir tester *souvent*, aussitôt que des changements significatifs ont été effectuésà un module ou au programme :
  - pour assurer que tout fonctionne bien malgré les changements effectués ;
  - pour mieux localiser la source du problème lorsque ça ne fonctionne pas : si on effectue les tests après avoir fait de nombreux ajouts ou modifications, alors il est plus difficile de savoir d'où vient le problème.
- Le plus tôt on trouve les problèmes et erreurs (les *bogues*), le plus facile il devient de les localiser et de les corriger ⇒

  "*Code a little, test a little*".
- Le travail de programmation et construction d'un composant logiciel n'est pas terminé tant que tous les tests appropriés n'ont pas été *définis et exécutés* :
  - Les différents modules réussissent correctement les tests unitaires associés.
  - Les différents composants et sous-systèmes réussissent correctement les tests (d'intégration, unitaire de haut niveau) associés.
  - Le système dans son ensemble réussit correctement les tests d'acceptation.
- Si une erreur (un *bogue*) est détectée une première fois, il ne faut surtout pas que cette même erreur réapparaisse ultérieurement. Donc, si une erreur est détectée, il est crucial qu'un cas d'essai approprié soit ajouté aux cas de test, et que ce cas de test soit ré-exécuté à chaque fois que des modifications sont apportées au système. [HT00]

[2] Certains auteurs disent plutôt «*tests de non régression*» : la comportement du logiciel n'a pas regressé suite aux modifications qui viennent d'être d'effectuées.

Les cadres d'exécution automatique des tests

De nos jours, il existe de nombreux outils qui permettent d'automatiser l'exécution des tests unitaires, qui aident à développer de tels tests et à les associer au code testé, et ce pour la plupart des langages de programmation. Ces outils sont appelés des *cadres de test*(*tests framework*). Le cadre de test le plus connu est `JUnit` [BG98, Bec01, HT03], popularisé par les promoteurs de l'approche XP (*eXtreme Programming*) [Bec00].

La caractéristique fondamentale de tous les cadres de test est la suivante :

On utilise des *assertions*, plutôt que de vérifier des résultats textuels.

$\Rightarrow$ Aucun résultat n'est produit si le test ne détecte pas d'erreur.

3.3 Tests unitaires avec JUnit 4.0 : Un aperçu

Dans ce qui suit, nous présenterons la nouvelle version de JUnit, à savoir JUnit 4.0, qui utilise le mécanisme des annotations de Java 5.0. Pour des informations additionnelles, voir la référence suivante :

- *An early look at JUnit 4*, de E. Harold,

  `http://www-128.ibm.com/developerworks/java/library/j-junit4.html`

La présentation détaillée de JUnit se fera par l'intermédiaire d'un exemple. Plus précisément, dans la prochaine section, nous allons développer un certain nombre de tests pour classes Java définissant des *ensembles génériques à la OCL* — faisant partie d'une bibliothèque que nous appellerons `OclCollections` — lesquelles représentent des collections de valeurs (type immuable).

JUnit est un *cadre de test*(*test framework*) — donc une bibliothèque qui définit un ensemble de classes *qu'on peut étendre* — qui facilite le développement des tests — de classes, groupes de classes ou applications.

Un *programme de test* est composé d'un ensemble de *classes de test*. Chaque classe de test est elle-même composée d'un ensemble de *méthodes de test* — on dit aussi qu'un programme de test est composé de *suites de test*, elles-mêmes composées de *cas de test*.

Le fonctionnement d'une méthode de test est relativement simple, sa signature étant restreinte à ne recevoir aucun argument et à ne produire aucun résultat — donc signature «`public void methodeDeTest()`». Ainsi, une méthode de test pour à tester une certaine méthode, en gros, fonctionne comme suit — ce que Meszaros [Mes07]) dénote par *Setup–Exercise–Verify–Teardown* :

- (*Setup*) On crée les objets requis pour mettre en oeuvre le cas de test.
- (*Exercise*) On appelle la méthode à tester (si nécessaire, en appellant aussi d'autres méthodes).
- (*Verify*) On vérifie que le résultat retourné ou l'effet produit par la méthode est bien celui désiré. Cette vérification se fait *à l'aide d'assertions*, donc sans émettre de résultats ou informations à l'extérieur (ou presque).
- (*Teardown*) On effectue le nettoyage des données ou effets (pour permettre l'exécution des autres cas de test).

Avant d'aborder les détails plus spécifiques, voyons la mise en oeuvre en Java d'ensembles génériques à la OCL.

4 Un type abstrait immuable Java pour des ensembles génériques style OCL : `OclCollections.Set`

Dans cette section, nous présentons la spécification (fichiers d'`interface`) et la mise en oeuvre (classes abstraites et classes concrètes) pour des ensembles génériques de style OCL d'une bibliothèque `OclCollections`. Cette bibliothèque définit aussi des sacs (`Bag`) aet des séquences (`Sequence`) génériques, mais leur mise en oeuvre ne sera pas discutée dans le présent document.

4.1 Les interfaces `Collection<T>` et `Set<T>`

Fichier Java 2 Interface pour un type `Collection`, semblable à celui d'OCL : `Collection.java`.

```
package OclCollections;
import java.util.Iterator;

public interface Collection<T> extends Iterable<T> {
    int size();
    boolean isEmpty();
    int count( T elem );

    boolean includes( T elem );
    boolean includesAll( Collection<T> col );

    boolean exists( Expression<T,Boolean> p );
    boolean forAll( Expression<T,Boolean> p );

    <R> R iterate( R initVal, BinaryExpression<T,R> expr );

    // Methodes specifiques a la mise en oeuvre Java.
    Iterator<T> iterator();

    boolean equals( Object s2 );

    String toString();
}
```

Le Fichier Java 2 présente une `interface` pour un type abstrait `Collection<T>` semblable au type `Collection` d'OCL. Soulignons tout d'abord que cette interface présente en fait un *sous-ensemble* des opérations permises sur ce type.[3] Nous ne traiterons ici qu'une partie de ce sous-ensemble des opérations, et ce de façon à alléger la présentation. La spécification complète de la bibliothèque `OclCollections` est disponible sur le site Web du cours et la mise en oeuvre réalise la quasi-totalité des opérations sur les types `Set`, `Bag` et `Sequence` : `http://www.info2.uqam.ca/~tremblay/INF3140/OclCollections`

Toujours de façon à alléger la présentation du présent document, les fichiers ne sont pas documentés (ou très peu) — comme on va le voir, les méthodes sont quasi-identiques aux opérations OCL, sauf pour la forme de certains appels.

Le Fichier Java 3 présente une `interface` pour un type abstrait `Set<T>` semblable au type `Set` d'OCL. Encore une fois, il s'agit en fait d'un sous-ensemble des opérations permises

[3] Les autres opérations, qui ne seront pas traitées ici, sont les suivantes : `notEmpty`, `excludes`, `excludesAll`, `one`, `isUnique`, `any`.

Fichier Java 3 Interface pour un type `Set`, semblable au type `set` d'OCL : `Set.java`.

```
package OclCollections;
import java.util.Iterator;

public interface Set<T> extends Collection<T> {
    Set<T> including( T elem );

    Set<T> union( Set<T> s2 );
    Set<T> intersection( Set<T> s2 );

    Set<T> minus( Set<T> s2 );

    Sequence<T> asSequence();

    Set<T> select( Expression<T,Boolean> p );
    Set<T> reject( Expression<T,Boolean> p );

    <R> Bag<R> collect( Expression<T,R> expr );
}
```

sur ce type.[4] On y remarque que cette interface hérite de l'interface `Collection` — `Set<T> extends Collection<T>`. Un objet de type `Set` est dont aussi de type `Collection`, et donc supporte *toutes les méthodes décrites dans ces deux (2) interfaces.*

Il s'agit ici, tant pour `Collection` que pour `Set`, de types *génériques*, chacun défini par une `interface` générique d'argument `T` — indiqué par "`<T>`". Puisqu'il s'agit d'une `interface`, on indique donc uniquement les méthodes qui s'appliquent *à un objet donné* — c'est-à-dire, uniquement les méthodes d'instances, *sans méthode statique* et *sans constructeur.*

Fichier Java 4 Interface `Iterator<T>` de la bibliothèque Java.

```
  public interface Iterator<T> {
    boolean hasNext();
    // Returns true if the iteration has more elements.

    T next();
    // Returns the next element in the iteration.

    void remove()
    // Removes from the underlying collection the last element
    // returned by the iterator (optional operation).
  }
```

Fichier Java 5 Interface `Iterable<T>` de la bibliothèque Java.

```
  public interface Iterable<T> {
    Iterator<T> iterator();
    // Returns an iterator over a set of elements of type T.
  }
```

On remarque aussi que l'interface `Collection` (Fichier Java 2) spécifie une opération `iterator()` qui retourne un objet de type `Iterator<T>`. Cette dernière interface est définie

[4] Dans ce cas, les opérations omises sont les suivantes : `excluding`,`symmetricDifference`, `asBag`.

telle qu'indiquée dans le Fichier Java 4 et permet d'obtenir *de façon itérative* les divers éléments d'une collection. (Voir la Parenthèse 1 pour plus de détails sur les `Iterator` et les boucles `for` en Java 5.0.) On remarque aussi que le fait d'avoir une opération `iterator()` qui permet d'obtenir un `Iterator<T>` est décrit par l'interface `Iterable<T>` (Fichier Java 5). Comme on constate avec l'interface `Collection<T>`, cette dernière met effectivement en oeuvre l'interface `Iterable<T>`.

Les boucles `for` et l'interface `Iterator`

Soit un ensemble `Set<Integer> s`. Supposons que l'on désire calculer la somme des éléments de `s`. En utilisant l'ancienne syntaxe de Java, on aurait écrit ce qui suit :

```
Iterator<Integer> it = s.iterator();
int total = 0;
while( it.hasNext() ) {
  total += it.next();
}
```

Par contre, avec la nouvelle syntaxe pour les boucles `for` (Java 5.0), on peut écrire une boucle équivalente de façon beaucoup plus simple, les appels aux diverses opérations de l'itérateur (`iterator`, `hasNext` et `next`) étant implicites (générées par le compilateur) :

```
int total = 0;
for( Integer i : s ) {
  total += i;
}
```

La syntaxe d'une telle boucle `for` introduit donc le type et le nom de la variable d'itération (`Integer` et `i`) suivi après le «:» d'une objet qui met en oeuvre l'interface `Iterable`, donc qui fournit une méthode `iterator` qui retourne un objet de type `Iterator`, possédant donc les méthodes `hasNext` et `next`.

On remarque aussi que bien que la variable `i` soit déclarée de type `Integer`, elle peut être utilisée directement dans l'addition à `total`, de type `int` : c'est ce que l'on appelle «*auto-boxing/unboxing*», c'est-à-dire la conversion implicite entre objet et type primitif, sans nécessité d'utiliser explicitement un *wrapper* — un comportement similaire survient aussi dans le premier exemple, puisque `it.next()` retourne aussi un `Integer`.

Parenthèse 1: Les nouvelles boucles `for` en Java 5.0 et le *auto-boxing/unboxing* des valeurs primitives.

4.2 Les classes abstraites `AbstractCollection<T>` et `AbstractSet<T>`

Les fichiers 6 et 7 présentent des extraits de deux classes abstraites : `AbstractCollection` et `AbstractSet`.

Une classe qui possède au moins une méthode abstraite (soit indiquée explicitement par `abstract`, soit qu'elle n'est pas définie parce qu'elle le sera dans une sous-classe) est une classe dite «abstraite» — qu'on indique alors comme étant telle à l'aide du mot-clé `abstract`. C'est bien le cas de ces deux classes, qui ne définissent pas la méthodes `iterator` et, dans le cas de `AbstractSet`, qui ne définit pas la méthode `including`.

Une particularité d'une classe abstraite est qu'elle ne peut pas être instanciée (on ne peut pas faire de `new`), puisque certaines méthodes ne sont pas définies. En d'autres mots, il est impossible de créer un objet d'une telle classe, qui ne peut pas posséder de constructeur.

Par contre, comme on le constate dans les deux cas, une classe abstraite peut quand mettre en oeuvre certaines opérations. En d'autres mots, plusieurs des méthodes sur les

Fichier Java 6 Classe abstraite `AbstractCollection` : `AbstractCollection.java`.

```
package OclCollections;
import java.util.Iterator;

abstract public class AbstractCollection<T> implements Collection<T>, Iterable<T> {

    public boolean includes( T elem ) {
       for( T e : this ) {
           if ( e.equals( elem ) ) { return true; }
       }
       return false;
    }

    public int size() {
       int nb = 0;
       for( T e : this ) {
           nb += 1;
       }
       return nb;
    }

    public boolean isEmpty() {
       return size() == 0;
    }

    public int count( T elem ) {
       int nb = 0;
       for( T e : this ) {
           if ( elem.equals(e) ) { nb += 1; }
       }
       return nb;
    }

    public boolean includesAll( Collection<T> s2 ) {
       for( T e : s2 ) {
           if ( !includes( e ) ) { return false; }
       }
       return true;
    }
}
```

Fichier Java 7 Classe abstraite `AbstractSet` : `AbstractSet.java`.

```
package OclCollections;
import java.util.Iterator;

abstract public class AbstractSet<T> extends AbstractCollection<T> implements Set<T>, Iterable<T> {

    public Set<T> union( Set<T> s2 ) {
       Set<T> u = s2;
       for( T e : this ) {
           u = u.including( e );
       }
       return u;
    }

    public Set<T> intersection( Set<T> s2 ) {
       Set<T> u = Collections.emptySet();
       for( T e : this ) {
           if ( s2.includes( e ) ) { u = u.including( e ); }
       }
       return u;
    }

    public Set<T> minus( Set<T> s2 ) {
       Set<T> u = Collections.emptySet();
       for( T e : this ) {
           if ( !s2.includes( e ) ) { u = u.including( e ); }
       }
       return u;
    }

    @Override
    @SuppressWarnings("unchecked")
    public boolean equals( Object o ) {
       Set<T> s;
       if (o instanceof Set<?>) {
           s = (Set<T>) o; // unchecked cast
       } else {
           return false;
       }
       if ( size() != s.size() ) { return false; }
       for ( T e : this ) {
           if ( !s.includes( e ) ) { return false; }
       }
       return true;
    }

    public Sequence<T> asSequence() {
       Sequence<T> res = Collections.emptySequence();
       for ( T e : this ) {
           res = res.prepend( e );
       }
       return res;
    }
}
```

collections ou les ensembles peuvent être définies directement dans la classe abstraite, de façon complètement indépendante de la mise en oeuvre dans une classe concrète — on parle alors de «*skeletal implementation*». Dans le cas présent, ceci est possible en utilisant l'`iterator` associé à la collection ou à l'ensemble, lequel permet de parcourir, et donc traiter/analyser, chacun des éléments de la collection ou de l'ensemble.

Pour créer et obtenir des objets concrets utilisables, il faut donc définir une ou plusieurs classes concrètes, qui fourniront des mises en oeuvre de chacune des méthodes abstraites et qui fourniront aussi des constructeurs. Dans ce qui suit, nous verrons deux classes concrètes pour des ensembles : `LinkedListSet` (section 4.3) et `ArraySet` (section 5.3).

4.3 La classe concrète `LinkedListSet<T>`

Tel que mentionné plus haut, la classe abstraite `AbstractSet` ne définit pas de constructeurs. Rappelons qu'il n'est jamais possible d'instancier une classe abstraite, donc un constructeur ne peut pas être défini pour une telle classe : il faut plutôt définir *une sous-classe*, *concrète*, qui elle pourra ultimement être instanciée. Cette sous-classe *devra* définir une mise en oeuvre pour les méthodes indiquées comme étant `abstract` — `including` et `iterator` — et *pourra* aussi *redéfinir* certaines des méthodes déjà définies dans la classe abstraite — `size` dans cet exemple. Le Fichier Java 8 présente une telle mise en oeuvre des ensembles par une classe concrète `LinkedList`, où les ensembles sont mis en oeuvre à l'aide de listes chaînées linéaires.

Dans ce qui suit, pour illustrer le fonctionnement du cadre de tests JUnit 5.0, nous allons supposer que les diverses classes (déjà compilées) définissant les types génériques pour collections et ensembles à la OCL ont été compilées combinées dans un fichier `jar` appelé `OclCollections.jar`.

5 Programmes de test JUnit 4.0 pour les ensembles génériques `OclCollections.Set`

Dans cette section, nous allons développer différents programmes de tests pour les ensembles génériques : une première version simple, écrite dans un style «*plus ou moins élégant*», une deuxième version avec un meilleur style, puis finalement une dernière version utilisant le patron de conception «*fabrique*» pour définir de façon simple des littéraux (constantes) de type ensemble.

Remarque sur la méthode `equals`

On remarque que la classe `AbstractSet` (fichier 7) définit la méthode `equals`. Il s'agit d'une redéfinition (dans une sous-classe) de la méthode applicable à n'importe quel objet — donc définie par défaut dans la classe `Object` — et c'est ce qui explique la signature de cette méthode (argument de type `Object`) et l'utilisation de l'annotation «`@Override`»

Une solution qui aurait semblé «naturelle» aurait été de définir `equals` comme suit, tant dans l'interface `Collection<T>` que dans les classes abstraites :

```
boolean equals( Collection<T> o );
```

Malheureusement, cette solution ne fonctionne pas si on veut avoir des collections hiérarchiques arbitraires, i.e., des collections dont les éléments sont des collections : la méthodes `equals` définie sur une collection utilise `includes` pour déterminer si un élément est présent ou non, laquelle méthode utilise à son tour `equals` pour comparer les éléments, de type arbitraire!

Remarque sur la méthode `hashCode`

Dans la documentation en ligne de Sun sur Java, on trouve la remarque suivante concernant la méthode `equals`[5] :

> *Note that it is generally necessary to override the* `hashCode` *method whenever this method [*`equals`*] is overridden, so as to maintain the general contract for the* `hashCode` *method, which states that equal objects must have equal hash codes.*

Le respect des contraintes additionnelles suivantes est aussi considéré comme important pour les méthodes `equals` et `hashCode`[6] :

- si deux objets sont égaux selon `equals`, leurs `hashCode` doivent être égaux ;
- si les méthodes `equals` et `hashCode` sont toutes deux redéfinies, elles devraient utiliser les mêmes champs ;
- si un objet est immuable, alors `hashCode` est une méthode candidate pour la mémorisation (*caching*) et l'initialisation paresseuse ;

Une mise en oeuvre possible de `hashCode` (dans `AbstractSet`) pourrait donc être la suivante :

```
private int hashCode;
private boolean hashCodeDefini = false; // Pour memorisation.

@Override
public int hashCode() {
  if ( ! hashCodeDefini ) { // Initialisation paresseuse et memorisation.
      hashCode = 0;
      for( T e : this ) {
        hashCode += e.hashCode();
      }
      hashCodeDefini = true;
  }
  return hashCode;
}
```

Signalons toutefois que pour cette méthode fonctionne correctement pour n'importe quelle collection, y compris celles de très grandes tailles, il serait préférable d'utiliser une addition modulo un nombre premier pour éviter les débordements.

Parenthèse 2: Les méthodes `equals` et `hashCode`.

Fichier Java 8 Classe pour une mise en oeuvre concrète des ensembles avec des listes linéaires chaînées : `LinkedListSet.java`.

```
package OclCollections;
import java.util.Iterator;

public final class LinkedListSet<T> extends AbstractSet<T> {

    // Variables d'instance: Mise en oeuvre avec liste chainee lineaire.
    private T                elem;         // L'element de l'ensemble.
    private LinkedListSet<T> prochain;     // Le prochain noeud de la liste.
    private int              nbElems;      // Le nombre d'elements de la sous-liste.


    // Constructeurs (prives).
    public LinkedListSet() {
       this( null, null, 0 );     // Noeud representant l'ensemble vide.
    }

    private LinkedListSet( T e, LinkedListSet<T> p, int n ) {
       elem = e;
       prochain = p;
       nbElems = n;
    }


    // Operations concretes pour les deux methodes abstraites.
    public Set<T> including( T elem ) {
       return includes(elem) ? this : new LinkedListSet<T>( elem, this, nbElems+1 );
    }

    public Iterator<T> iterator() {
       return new SetIterator<T>();
    }

    // Redefinition d'une methode abstraite par une autre methode concrete.
    @Override
    public int size() {
       return nbElems;
    }

    // Classe auxiliaire pour l'iterateur.
    @SuppressWarnings("unchecked")
    class SetIterator<T> implements Iterator<T> {
       LinkedListSet<T> prochainElem;

       public SetIterator() {
           prochainElem = (LinkedListSet<T>) LinkedListSet.this; // unchecked cast
       }

       public boolean hasNext() {
           return prochainElem.prochain != null;
       }

       public T next() {
           T e = prochainElem.elem;
           prochainElem = prochainElem.prochain;
           return e;
       }

       public void remove() throws UnsupportedOperationException {
           throw new UnsupportedOperationException();
       }
    }
}
```

5.1 Un premier programme de test

Fichier Java 9 Classe de test JUnit pour les ensembles génériques : `ExempleTesterSet1.java`.

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.Iterator;
import OclCollections.*;
import static OclCollections.Collections.*;

public class ExempleTesterSet1 {

    @Test public void zero_element() {
       Set<Integer> vide = new LinkedListSet<Integer>();

       assertEquals( 0, vide.size() );

       assertEquals( vide, vide );
       assertEquals( vide, new LinkedListSet<Integer>() );
       assertFalse ( vide.equals( new LinkedListSet<Integer>().including(1) ) );

       assertFalse( vide.includes(0) );
       assertFalse( vide.includes(1) );
    }

    @Test public void un_element() {
       Set<Integer> s1 = new LinkedListSet<Integer>().including(1);

       assertEquals( 1, s1.size() );

       assertEquals( s1, s1 );
       assertEquals( s1, new LinkedListSet<Integer>().including(1).including(1) );
       assertFalse ( s1.equals( new LinkedListSet<Integer>() ) );

       assertFalse( s1.includes( 0 ) );
       assertTrue ( s1.includes( 1 ) );
       assertFalse( s1.includes( 2 ) );
       assertFalse( s1.includes( 3 ) );
    }

    @Test public void deux_elements() {
       Set<Integer> s12 = new LinkedListSet<Integer>().including(1).including(2);

       assertEquals( 2, s12.size() );

       assertEquals( s12, s12 );
       assertEquals( s12, new LinkedListSet<Integer>().including(2).including(1).including(2) );
       assertFalse ( s12.equals( new LinkedListSet<Integer>().including(1) ) );

       assertFalse( s12.includes( 0 ) );
       assertTrue ( s12.includes( 1 ) );
       assertTrue ( s12.includes( 2 ) );
       assertFalse( s12.includes( 3 ) );
       assertFalse( s12.includes( 4 ) );
    }
}
```

Le Fichier Java 9 présente une première version d'un programme de test pour le type générique `Set<T>` et les classes `AbstractSet<T>` et `LinkedListSet<T>` associées :

- Ce programme de test définit trois (3) méthodes de test — trois cas de test. Chaque méthode de test — de type `void _()`, donc ne recevant aucun argument et ne retournant

aucun résultat — est identifiée à l'aide d'une annotation «`@Test`».

- La forme générale d'une méthode de test est la suivante :
  - Construire un (ou plusieurs) objet(s).
  - Effectuer diverses opérations sur l'objet.
  - Vérifier que l'objet résultant satisfait diverses propriétés attendues, et ce en utilisant des assertions.

Pour illustrer l'utilisation de JUnit, nous allons supposer (voir plus haut) que les classes définissant les ensembles génériques sont dans le fichier `OclCollections.jar`.

On peut alors compiler et exécuter le programme de test indiqué dans le Fichier Java 9 (classe `ExempleTesterSet1`) à l'aide des commandes suivantes, ce qui génère les résultats d'exécution suivants (le «\» ne fait pas partie de la commande mais indique simplement une continuation de ligne) :

```
% javac -cp OclCollections.jar:/usr/local/opt/junit/junit.jar ExempleTesterSet1.java
% java  -ea -cp .:OclCollections.jar:/usr/local/opt/junit/junit.jar  \
                org.junit.runner.JUnitCore  ExempleTesterSet1
JUnit version 4.4
...
Time: 0,144

OK (3 tests)
```

L'appel à la machine virtuelle Java exécute donc en fait le programme `org.junit.runner.JUnitCore` avec comme argument, sur la ligne de commande, la classe de test `ExempleTesterSet1`.[7] Comme on le voit pour cet exemple d'exécution, trois (3) tests ont été exécutés, tous avec succès — un «.» dénote l'exécution réussie d'une méthode de test.

[7] Il est aussi possible d'indiquer plusieurs arguments lors de l'appel à `org.junit.runner.JUnitCore`, i.e., plusieurs classes de test.

```
% java -ea -cp .:OclCollections.jar:/usr/local/opt/junit/junit.jar  \
                org.junit.runner.JUnitCore  ExempleTesterSet1
JUnit version 4.4
...E
Time: 0,161
There was 1 failure:
1) deux_elements(ExempleTesterSet1)
java.lang.AssertionError:
        at org.junit.Assert.fail(Assert.java:74)
        at org.junit.Assert.assertTrue(Assert.java:37)
        at org.junit.Assert.assertFalse(Assert.java:56)
        at org.junit.Assert.assertFalse(Assert.java:65)
        at ExempleTesterSet1.deux_elements(ExempleTesterSet1.java:43)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:585)
        at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:59)
        at org.junit.internal.runners.MethodRoadie.runTestMethod(MethodRoadie.java:98)
        at org.junit.internal.runners.MethodRoadie$2.run(MethodRoadie.java:79)
        at org.junit.internal.runners.MethodRoadie.runBeforesThenTestThenAfters(MethodRoadie.java:87)
        at org.junit.internal.runners.MethodRoadie.runTest(MethodRoadie.java:77)
        at org.junit.internal.runners.MethodRoadie.run(MethodRoadie.java:42)
        at org.junit.internal.runners.JUnit4ClassRunner.invokeTestMethod(JUnit4ClassRunner.java:88)
        at org.junit.internal.runners.JUnit4ClassRunner.runMethods(JUnit4ClassRunner.java:51)
        at org.junit.internal.runners.JUnit4ClassRunner$1.run(JUnit4ClassRunner.java:44)
        at org.junit.internal.runners.ClassRoadie.runUnprotected(ClassRoadie.java:27)
        at org.junit.internal.runners.ClassRoadie.runProtected(ClassRoadie.java:37)
        at org.junit.internal.runners.JUnit4ClassRunner.run(JUnit4ClassRunner.java:42)
        at org.junit.internal.runners.CompositeRunner.runChildren(CompositeRunner.java:33)
        at org.junit.internal.runners.CompositeRunner.run(CompositeRunner.java:28)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:130)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:109)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:100)
        at org.junit.runner.JUnitCore.runMain(JUnitCore.java:81)
        at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

FAILURES!!!
Tests run: 3,  Failures: 1
```

Figure 2: Exemple d'une méthode de test qui échoue.

Supposons plutôt que, à cause d'une erreur de programmation (style erreur de *cut-and-paste*), la dernière ligne du test `deux_elements` soit plutôt l'instruction suivante :

```
assertFalse( s12.includes( 2 ) );
```

La Figure 2 illustre alors le résultat de l'exécution de la classe de test — un «E» indique l'échec d'une méthode de test, c'est-à-dire, la présence d'un cas de test contenant une assertion erronée, la localisation exacte de cette erreur étant ensuite donnée plus en détail par l'impression de la pile d'exécution au point où l'assertion invalide a été détectée.

5.2 Une version révisée du programme de test : Application du principe «*DRY*»

Les règles d'écriture d'un programme de test sont les mêmes que celles pour un programme «normal» — qui réalise les fonctionalités du logiciel. Un principe important d'un bon style de programmation est le principe *DRY — Don't Repeat Yourself* [HT00] — qui dit que si du code identique (ou presque) se repète à différents endroits, c'est un signe que le code peut être amélioré en introduisant, par exemple, des variables ou méthodes auxiliaires. L'exemple précédent ne respecte pas bien cette règle car certains ensembles qui réapparaissent à différents

L'instruction Java `assert` vs. l'assertion JUnit `assertTrue`

À première vue, la signification de l'instruction `assert` du langage Java et celle de l'assertion `assertTrue` du cadre de tests JUnit semblent identiques : si l'expression indiquée en argument s'évalue à `true`, alors rien ne se passe ; sinon, une «erreur» survient. Il ne faut toutefois pas confondre les deux instructions, le contexte d'exécution et l'effet produit étant tout à fait différents :

`assert e;`

- Cette instruction peut être utilisée dans n'importe quelle classe Java.
- Si l'expression `e` s'évalue à `false` et que l'option `-ea` n'est pas activée, alors aucun effet.
- Si l'expression `e` s'évalue à `false` et que l'option `-ea` est activée, alors l'exception `java.lang.AssertionError` est signalée — ce qui termine l'exécution du programme si elle n'est pas traitée, ce qui devrait être le comportement normal.

`assertTrue( e );`

- Cette instruction ne peut être utilisée que dans une classe Java qui importe les méthodes de la bibliothèque JUnit et qui est ensuite exécutée par un `runner` de cette bibliothèque.
- Si l'expression `e` s'évalue à `false`, alors un message d'échec de tests (*failure*) sera inclus dans le rapport d'exécution des tests, que l'option `-ea` soit activée ou non, et l'exécution se poursuit ensuite avec le test suivant.

Parenthèse 3: L'instruction Java `assert` vs. l'assertion JUnit `assertTrue`.

endroits sont réécrits de façon explicite. Pour éviter de telles répétitions, il est possible d'introduire des variables privées mais partagées par les différentes méthodes de test, variables qui sont initialisées juste avant l'exécution de chacun des tests — soulignons que chaque test doit être conçu pour s'exécuter de façon indépendante des autres tests, puisque l'ordre dans lequel les différentes méthodes de test s'exécutent n'est pas défini.

Le Fichier Java 10 illustre l'utilisation de telles variables partagées, lesquelles sont initialisées par la méthode `definirConstantes`.[8] On remarque que cette méthode est annotée avec «`@Before`», ce qui signifie qu'elle doit être exécutée juste avant l'exécution de chacun des tests. On remarque aussi que des méthodes auxiliaires ont été introduites pour permettre de vérifier, toujours à l'aide d'assertions, qu'un certain intervalle de valeurs est présent, ou absent, d'un ensemble — méthodes `assertAllIn` et `assertNoneIn` — ce qui permet d'éviter de répéter de nombreux `assertTrue` ou `assertFalse` pour vérifier la présence, ou non, d'un certain nombre d'éléments.

5.3 Une autre version du programme de test : Utilisation de méthodes de *fabrication*

Comme on l'a vu dans le Fichier Java 10, la notation Java permet d'utiliser des appels de méthodes qui ressemblent assez à ceux de OCL — `s.including(1)`, `s.size()`. Par contre, il n'est pas possible d'écrire directement des littéraux comme en OCL — i.e. `Set{1, 2, 3}`. Il est toutefois possible de s'approcher de cette notation en utilisant un nouvel élément de Java 5.0, à savoir les méthodes avec un nombre variable d'arguments.

[8] Les `import` au début du fichier ont été omis, faute de place. Ils sont exactement les mêmes que dans le Fichier Java 9.

Fichier Java 10 Version révisée de la classe de test JUnit pour les ensembles génériques: `ExempleTesterSet2.java`.

```
public class ExempleTesterSet2 {

    // Methode auxiliaire pour creer un "nouvel" ensemble vide.
    private Set<Integer> mkEmpty() { return( new LinkedListSet<Integer>() ); }

    // Constante pour un ensemble vide.
    final private Set<Integer> EMPTY = mkEmpty();

    // Variables partagees entre les differents tests.
    private Set<Integer> vide, s1, s12;

    // Methodes auxiliaires d'assertions.
    private void assertAllIn( Set<Integer> s, int inf, int sup ) {
       for( int i = inf; i <= sup; i++ ) {
           assertTrue( s.includes( i ) );
       }
    }

    private void assertNoneIn( Set<Integer> s, int inf, int sup ) {
       for( int i = inf; i <= sup; i++ ) {
           assertFalse( s.includes( i ) );
       }
    }

    // Methode pour effectuer le "setup" de la "fixture" de test.
    @Before public void definirConstantes() {
       vide  = EMPTY;
       s1    = EMPTY.including(1);
       s12   = EMPTY.including(2).including(1);
    }

    @Test public void zero_element() {
       assertEquals( 0, vide.size() );

       assertEquals( vide, vide );
       assertEquals( vide, mkEmpty() );
       assertFalse ( vide.equals(s1) );

       assertNoneIn( vide, 0, 1 );
    }

    @Test public void un_element() {
       assertEquals( 1, s1.size() );

       assertEquals( s1, s1 );
       assertEquals( s1, EMPTY.including(1).including(1) );
       assertFalse ( s1.equals( vide ) );

       assertNoneIn( s1, 0, 0 );
       assertAllIn ( s1, 1, 1 );
       assertNoneIn( s1, 2, 3 );
    }

    @Test public void deux_elements() {
       assertEquals( 2, s12.size() );

       assertEquals( s12, s12 );
       assertEquals( s12, EMPTY.including(2).including(1).including(2) );
       assertFalse ( s12.equals(s1) );

       assertNoneIn( s12, 0, 0 );
       assertAllIn ( s12,  1, 2 );
       assertNoneIn( s12,  3, 4 );
    }
}
```

Dans ce qui suit, nous allons définir deux méthodes permettant de créer des ensembles, dont une sera une méthode *statique* `mkSet` déclarée et définie à l'aide d'un nombre variable d'arguments provenant d'une classe `Collections`. Des extraits de cette classe sont présentées dans le fichier Java 11. Les parenthèses 4 et 5, quant à elles, expliquent le principe de base des méthodes avec un nombre variable d'arguments.

Les méthodes avec un nombre variable d'arguments

Une méthode avec un paramètre formel `elems` déclarée avec «`T... elems`» indique que `elems` représente en fait *un nombre variable d'éléments* de type `T`. La méthode peut donc recevoir 0, 1 ou plusieurs arguments de type `T`, les arguments effectifs pouvant être obtenus à l'aide d'une boucle `for` — soit via un `iterator`, soit via des index pour un tableau, les arguments effectifs étant en fait transmis dans un tableau.

Par exemple, les méthodes suivantes permettent d'effectuer la somme, resp. le produit, des divers entiers reçus en argument :

```
static public Integer somme( Integer... args ) {
   int total = 0;
   for( Integer x: args ) {
       total += x;
   }
   return total;
}

static public Integer produit( Integer... args ) {
   int produit = 1;
   for( int i = 0; i < args.length; i++ ) {
       produit *= args[i];
   }
   return produit;
}
```

On peut donc effectuer des appels comme les suivants :

```
x = somme( 1, 2, 3, 4 );    // => 10

x = produit( 1, 2, 3, 4 );  // => 24

x = produit( 23 )           // => 23

x = produit();              // =>  1

x = somme();                // =>  0
```

Parenthèse 4: Méthodes avec un nombre variable d'arguments.

Plus précisément, deux méthodes statiques de constructions d'ensembles sont définies par la classe `Collections` :

- `emptySet()` : Retourne un ensemble vide.
- `mkSet()` : Retourne un ensemble contenant les divers éléments spécifiés en argument.

Une particularité de ces *méthode de fabrication* (*factory methods*) est qu'il est possible de choisir *entre deux mises en oeuvre possibles des ensembles* : des ensembles mis en oeuvre avec des listes chaînées (Fichier Java 8, p. 14) ou des ensembles mis en oeuvre avec des tableaux (voir classe `ArraySet<T>`, Fichier Java 12, p. 24).

Les méthodes avec un nombre variable d'arguments (suite)

Une méthode avec un nombre variable d'arguments peut être vue comme une méthode qui reçoit un tableau de valeurs du type indiqué. Donc, soit la déclaration et l'appel suivants :

```
Set<T> mkSet( T... elems ){
   /* Code omis */
}

... mkSet( 10, 20, 30 );
```

Ce code est alors équivalent à ce qui suit :

```
Set<T> mkSet( T[] elems ){
   /* Code omis */
}

...  mkSet( new T[] { 10, 20, 30 } );
```

C'est donc ce qui explique que dans une méthode avec un nombre variable d'arguments, où les arguments sont dénotés par le nom `elems`, on peut utiliser les expressions suivantes :

- `elems.length` : Pour déterminer le nombre d'arguments reçus lors de l'appel.
- `elems[i]` : Pour obtenir le ième argument.

Il peut être utile de souligner un *problème* avec de telles méthodes, lié à la façon dont les génériques sont traités en Java (*type erasure*) : lorsque les arguments transmis à une méthode avec un nombre variable d'arguments sont eux-mêmes des objets génériques — par exemple, un appel «`mkSet( s0 );`» où `s0` est lui-même un ensemble générique —, alors le compilateur va émettre un avertissement tel que le suivant :

```
TestSet.java:545: warning: [unchecked] unchecked generic array creation of type
    OclCollections.Set<java.lang.Integer>[] for varargs parameter
        Set<Set<Integer>> s2 = Collections.mkSet( s0 );
```

On peut alors supprimer cet avertissement en ajoutant l'annotation suivante devant la déclaration de la méthode contenant cet appel :

```
    @SuppressWarnings("unchecked")
```

Parenthèse 5: Méthodes avec un nombre variable d'arguments (suite).

Fichier Java 11 Fabriques pour construire des `Sets` : Extraits du fichier `Collections.java`.

```
package OclCollections;

public class Collections {
    /* Set. */

    /** Les deux (2) mises en oeuvre possibles pour les valeurs de type set. */
    public enum SetImplementation { LINKED_LIST, ARRAY };

    /** Pour definir la mise en oeuvre par defaut. */
    public static final void defineSetImplementation( SetImplementation impl ) {
       setImplementation = impl;
    }

    private static Set emptyLinkedListSet, emptyArraySet;
    private static SetImplementation setImplementation = SetImplementation.LINKED_LIST;

    // Constructeur statique pour un ensemble vide. L'objet retourne
    // est unique (le meme pour tous les appels) pour une mise en oeuvre donnee.
    @SuppressWarnings("unchecked")
    public static final <T> Set<T> emptySet() {
       switch( setImplementation ) {
       case LINKED_LIST:
           if ( emptyLinkedListSet == null ) { emptyLinkedListSet = new LinkedListSet<T>(); }
           return emptyLinkedListSet;
       case ARRAY:
           if ( emptyArraySet == null ) { emptyArraySet = new ArraySet<T>(); }
           return emptyArraySet;
       default:
           assert false : "Mise en oeuvre par defaut des ensembles invalides";
       }
       return null; // Pour eviter erreur du compilateur.
    }

    // Constructeur statique avec arguments multiples.
    public static final <T> Set<T> mkSet( T... elems ) {
       Set<T> s = emptySet();
       for( T e: elems ) {
           s = s.including( e );
       }
       return s;
    }

    /* Bag. */
    public static final <T> Bag<T> emptyBag() {
      // ...
    }

    public static final <T> Bag<T> mkBag( T... elems ) {
      // ...
    }

    /* Sequence. */
    public static final <T> Sequence<T> emptySequence() {
      // ...
    }

    public static final <T> Sequence<T> mkSequence( T... elems ) {
      // ...
    }

    /* Undefined. */
    public static Object Undefined = new Object();
}
```

Pour créer des ensembles avec une mise en oeuvre particulière, il faut préalablement faire un appel à la méthode `defineSetImplementation()`. Une fois cela fait, les appels subséquents à `emptySet()` ou `mkSet()` retourneront des ensembles mis en oeuvre avec la classe appropriée.

Comme l'illustre le Fichier Java 13 — classe de test `ExempleTesterSet3` — en utilisant la méthode `mkSet`, on peut alors écrire beaucoup plus simplement des valeurs littérales représentant des ensembles, tel que «`mkSet(1, 2, 3, 4)`», dont la forme est alors très semblable à celle du littéral OCL «`Set{1, 2, 3, 4}`». Voir aussi la parenthèse 6 pour des exemples additionnels.

On constate aussi — méthode de test `quatre_elements` — que deux ensembles peuvent être mis en oeuvre de façon différente (liste chaînée vs. tableau) tout en pouvant malgré tout être comparés correctement entre eux — ceci est possible grâce au fait que la méthode `equals` est définie dans la classes abstraite `AbstractSet` (mise en oeuvre non indiquée dans l'extrait fourni), donc ne dépend aucunement de la mise en oeuvre des ensembles comparés et ce grâce à l'utilisation de l'`iterator`.

Immutabilité des «objets» ⇒ Collection de «valeurs»

Les méthodes de test `immuable_un_meme_element` et `immuable_plusieurs_elements` du Fichier Java 13 (classe `ExempleTesterSet3`) permettent d'illustrer une propriété cruciale des ensembles à la OCL, propriété reflétée fidèlement dans la mise en oeuvre Java que nous avons présentée : un ensemble est une *entité immuable*, donc qui ne change jamais une fois créée. En Java, on parle alors d'«*objet immuable*» (*immutable object*)[9] ; par contre, comme on le verra par la suite, on parlera plus simplement de *valeur*, un objet étant considéré comme nécessairement mutable. Pour des exemples additionnels, et une comparaison avec les collections d'objets mutables provenant de la bibliothèque `java.util`, voir aussi la Section 7 (p. 34) ainsi que l'appendice E (p .43).

6 Mise en oeuvre des quantificateurs et des opérations générant des collections

6.1 La notion d'`Expression`

Pour mettre en oeuvre, en Java, des quantificateurs sur des collections, il nous faut rappeler qu'en OCL la forme générale d'un tel quantificateur est la suivante (idem pour `select` et `reject`) :

```
col->forAll( v | expression booléenne utilisant v )
col->exists( v | expression booléenne utilisant v )
```

Il nous faut donc, en Java, modéliser la notion «*d'expression booléenne utilisant* `v`»(où `v` est un élément de la collection). Toutefois, pour réaliser l'opération `collect`, il nous faudra aussi une notion plus générale d'expression, puisque la forme d'un appel à `collect` est la suivante, un tel appel générant une collection dont les éléments sont de type `R` :

```
col->collect( v | expression utilisant v et produisant une valeur de type R )
```

[9] Certains auteurs utilisent aussi le terme de «*value object*» [Bec07].

Fichier Java 12 Mise en oeuvre des ensembles avec des tableaux : `ArraySet.java`

```
package OclCollections;
import java.util.Iterator;

public final class ArraySet<T> extends AbstractSet<T> {

    // Variables d'instance: Mise en oeuvre avec tableau;
    final private int MAX_ELEMENTS = 1024;
    private int       nbElems;
    private T[]       elems;


    // Constructeurs (prives).
    @SuppressWarnings("unchecked")
    public ArraySet() {
       nbElems = 0;
       elems = (T[]) new Object[MAX_ELEMENTS];
    }

    //
    // Operations publiques concretes pour les deux methodes abstraites.
    //

    public Set<T> including( T elem ) {
       if ( includes(elem) ) {
           return this;
       } else {
           ArraySet<T> copie = new ArraySet<T>();
           for( int i = 0; i < nbElems; i++ ) {
              copie.elems[i] = elems[i];
              copie.nbElems += 1;
           }
           copie.elems[nbElems] = elem;
           copie.nbElems += 1;
           return copie;
       }
    }

    public Iterator<T> iterator() {
       return new Iterator<T>() {
           int prochainElem = 0;

           public boolean hasNext() {
              return prochainElem < ArraySet.this.nbElems;
           }

           public T next() {
              T e = (T) ArraySet.this.elems[prochainElem];
              prochainElem += 1;
              return e;
           }

           public void remove() throws UnsupportedOperationException {
              throw new UnsupportedOperationException();
           }
       };
    }

    public int size() {
       return nbElems;
    }
}
```

Fichier Java 13 Autre programme de test JUnit pour les ensembles génériques : `ExempleTesterSet3.java`.

```
public class ExempleTesterSet3 {

    // Variables partagees entre les differents tests.
    private Set<Integer> vide, s1, s12, s123, s1234;

    // Methodes auxiliaires d'assertions.
    private void assertAllIn( Set<Integer> s, int inf, int sup ) {
       // ...
    }

    private void assertNoneIn( Set<Integer> s, int inf, int sup ) {
       // ...
    }

    // Methode effectuer le "setup" de la "fixture" de test.
    @Before public void definirConstantes() {
       vide  = emptySet();
       s1    = mkSet( 1 );
       s12   = mkSet( 1, 2 );
       s123  = mkSet( 1, 2, 3 );
       s1234 = mkSet( 1, 2, 3, 4 );
    }

    @Test public void quatre_elements() {
       assertEquals( 4, s1234.size() );

       assertNoneIn( s1234, -2, 0 );
       assertAllIn ( s1234,  1, 4 );
       assertNoneIn( s1234,  5, 7 );

       Collections.defineSetImplementation( Collections.SetImplementation.ARRAY );
       Set<Integer> s1234_a = mkSet( 1, 2, 3, 4 );
       assertEquals( s1234, s1234_a );
       assertEquals( LinkedListSet.class, s1234.getClass() );
       assertEquals( ArraySet.class, s1234_a.getClass() );
    }

    @Test public void immuable_un_meme_element() {
      s1.including(2);

      assertEquals( mkSet(1), s1 );  // s1 est inchange.
      assertEquals( emptySet().including(1), s1 );

      assertNoneIn( s1, -1, 0 );
      assertAllIn ( s1,  1, 1 );
      assertNoneIn( s1,  2, 3 );

      s1.including(2);
      assertEquals( mkSet(1), s1 );  // s1 est inchange.
    }

    @Test public void immuable_plusieurs_elements() {
       s1.including(2);

       assertEquals( 1, s1.size() );
       assertEquals( mkSet(1), s1 );

       assertEquals( s12, s1.including(2).including(2).including(2) );

       assertEquals( 1, s1.size() );
       assertEquals( mkSet(1), s1 );  // s1 est inchange.

       assertEquals( 2, s12.size() );
       assertEquals( mkSet(1, 2), s12 );
    }
}
```

Similitudes/différences entre les valeurs en OCL vs. Java

Voici quelques exemples illustrant les similitudes/différences entre le style OCL et le style Java pour créer des valeurs littérales de type ensemble :

- `s: Set(Integer) = oclEmpty(Set(Integer))`

  ⇒

  ```
  Set<Integer> s = emptySet();
  ```

- `s: Bag(Integer) = Bag{}`

  ⇒

  ```
  Bag<Integer> s = emptyBag();
  Bag<Integer> s = mkBag();
  ```

- `s: Set(Integer) = Set{1..5}`

  ⇒

  ```
  Set<Integer> s = mkSet( 1, 2, 3, 4, 5 );
  ```

- `s: Set(String) = Set{'abc', 'def', ''}`

  ⇒

  ```
  Set<String> s = mkSet( "abc", "def", "" );
  ```

Parenthèse 6: Exemples illustrant les similitudes/différences entre les valeurs OCL vs. Java.

Nous allons donc définir la signature des méthodes `exists` et `forAll` comme suit :

```
boolean exists( Expression<T,Boolean> p );
boolean forAll( Expression<T,Boolean> p );
```

Quant aux méthodes qui produisent une collection à partir d'un ensemble, leur signature sera la suivante :

```
Set<T> select( Expression<T,Boolean> p );
Set<T> reject( Expression<T,Boolean> p );

<R> Bag<R> collect( Expression<T,R> expr );
```

Fichier Java 14 Interface pour des expressions : Fichier `Expression.java`.

```
package OclCollections;

public interface Expression<T,R> {
   public R eval( T v );
}
```

Ces méthodes font toute référence à une interface nommée `Expression`, définie dans le fichier Java 14. Un objet satisfait l'interface `Expression<T,R>` s'il possède une méthode `eval` qui, lorsqu'elle reçoit un objet `v` de type `T`, retourne une valeur de type `R` produite à partir de `v`.

Pour les quantificateurs et les méthodes `select` et `reject`, le type du résultat doit être un `Boolean` — on applique un *prédicat* sur chacun des éléments de la collection ou de l'ensemble — d'où l'utilisation de `Expression<T,Boolean>`.

Par contre, pour la méthode `collect`, qui permet de produire une nouvelle collection à partir de la collection de départ, collection résultante dont le type des éléments peut être distinct de celui de la collection de départ, on utilise plutôt `Expression<T,R>`, `T` représentant le type des éléments de la collection de départ et `R` le type des éléments de la collection résultante.

Fichier Java 15 Exemples d'objets de type `Expression` : `TestExpression.java`.

```
public class TestExpression {

    Expression<Integer,Boolean> egalA2 = new Expression<Integer,Boolean>() {
       public Boolean eval( Integer x ) {
            return x == 2;
       }
    };

    Expression<Integer,Boolean> divisiblePar2 = new Expression<Integer,Boolean>() {
       public Boolean eval( Integer x ) {
            return x % 2 == 0;
       }
    };

    Expression<Integer,Integer> fois3 = new Expression<Integer,Integer>() {
       public Integer eval( Integer x ) {
            return 3 * x;
       }
    };


    Expression<Integer,String> toString = new Expression<Integer,String>() {
       public String eval( Integer x ) {
            return x.toString();
       }
    };


    @Test public void egal_a_2() {
       assertTrue ( egalA2.eval(2) );
       assertFalse( egalA2.eval(3) );
    }

    @Test public void divisible_par_2() {
       assertTrue ( divisiblePar2.eval(2) );
       assertFalse( divisiblePar2.eval(3) );
    }

    @Test public void fois_3() {
       assertEquals( (Integer) 6, fois3.eval(2) );
       assertEquals( (Integer) 9, fois3.eval(3) );
    }
    @Test public void to_string() {
       assertEquals( "2", toString.eval(2) );
       assertEquals( "3", toString.eval(3) );
    }
}
```

Le fichier Java 15 présente quelques exemples de tels objets et leur utilisation, et ce à l'aide d'une classe de tests JUnit (`TestExpression`). Quant à la Parenthèse 7, elle explique l'utilisation des classes internes anonymes (*anonymous inner classes*), comme on en retrouve dans la fichier Java 15.

6.2 Mise en oeuvre des quantificateurs `exists` et `forAll`

Extrait de code Java 1 Mise en oeuvre des quantificateurs `exists` et `forall` dans la classe abstraite `AbstractCollection<T>`.

```
public boolean exists( Expression<T,Boolean> p ) {
   for ( T e : this ) {
       if ( p.eval(e) ) { return true; }
   }
   return false;
}

public boolean forAll( Expression<T,Boolean> p ) {
   for ( T e : this ) {
       if ( !p.eval(e) ) { return false; }
   }
   return true;
}
```

Dans la classe abstraite `AbstractSet<T>`, les opérations `exists` et `forAll` sont mises en oeuvre tel qu'indiqué dans l'Extrait de code Java 1 — l'argument de type `Expression<T,Boolean>` est nommé `p` comme rappel qu'il s'agit d'un prédicat.

Extrait de code Java 2 Exemples d'utilisation des quantificateurs `exists` et `forAll` sur des ensembles.

```
@Test public void exemple_quantificateurs() {
   Set<Integer> s0 = mkSet( 10, 20, 30 );

   assertTrue( s0.forAll( new Expression<Integer,Boolean>() {
                 public Boolean eval( Integer n ) {
                     return n % 2 == 0;
                 }
               } ) );


   assertTrue( s0.exists( new Expression<Integer,Boolean>() {
                 public Boolean eval( Integer n ) {
                    return n >= 30;
                 }
               } ) );
}
```

L'extrait de code Java 2 (p. 29) présente une méthode de test qui exprime, pour un ensemble `s0 = {10, 20, 30}`, les deux propriétés OCL suivantes :

```
s0->forAll( n | n.mod(2) = 0 )

s0->exists( n | n >= 30 )
```

Dans les deux cas, on constate qu'on y crée dynamiquement deux objets satisfaisant l'interface `Expression<T,Boolean>` — à l'aide de classes internes anonymes : voir Parenthèse 7 — et qu'on utilise ensuite ces objets dans les appels aux quantificateurs `exists` et `forAll`.

Les classes internes anonymes

Une interface ne peut jamais être instanciée : on ne peut jamais faire un `new` sur une `interface`, puisqu'une interface ne peut pas possèder pas de constructeur — une interface spécifie les méthodes d'instance, et un constructeur est en fait une méthode de classe.

Dans l'exemple 2, on remarque toutefois que certaines instructions font un appel à `new`, par exemple :

```
Expression<Integer,Boolean> egalA2 = new Expression<Integer,Boolean>() {
   public Boolean eval( Integer x ) {
        return x == 2;
   }
};
```

Contrairement à ce qu'on pourrait croire, l'objet obtenu n'est pas de la classe `Expression`, puisqu'`Expression` n'est pas une classe mais une interface. L'objet obtenu est plutôt une instance d'une *classe anonyme qui satisfait les contraintes de l'interface* `Expression` : ici, la seule contrainte d'un tel objet est de fournir une méthode `Boolean eval(Integer)`, et c'est bien une telle méthode qui est spécifiée dans l'extrait ci-haut.

Comme l'indique Flanagan [Fla02] :

> *An anonymous class is a local class without a name. An anonymous class is defined and instantiated in a single succinct expression using the new operator. While a local class definition is a statement in a block of Java code, an anonymous class definition is an expression, which means that it can be included as part of a larger expression, such as a method call. When a local class is used only once, consider using anonymous class syntax, which places the definition and use of the class in exactly the same place.*

Lorsqu'on compile un programme Java qui crée de telles classes, on peut remarquer que des classes auxiliaires sont effectivement créées. Par exemple, après la compilation du fichier Java 15 (classe `TestExpression`), on peut remarquer que les classes suivantes ont été générées :

```
$ ls *class
TestExpression.class    TestExpression$2.class  TestExpression$4.class
TestExpression$1.class  TestExpression$3.class
```

Parenthèse 7: Les classes internes anonymes.

6.3 Mise en oeuvre de `select`, `reject` et `collect`

Extrait de code Java 3 Mise en oeuvre des opérations `select`, `reject` et `collect` dans la classe abstraite `AbstractSet<T>`.

```
public Set<T> select( Expression<T,Boolean> p ) {
   Set<T> res = Collections.emptySet();
   for ( T e : this ) {
       if ( p.eval(e) ) { res = res.including( e ); }
   }
   return res;
}

public Set<T> reject( Expression<T,Boolean> p ) {
   Set<T> res = Collections.emptySet();
   for ( T e : this ) {
       if ( !p.eval(e) ) { res = res.including( e ); }
   }
   return res;
}

public <R> Bag<R> collect( Expression<T,R> expr ) {
   Bag<R> res = Collections.emptyBag();
   for ( T e : this ) {
       res = res.including( expr.eval(e) );
   }
   return res;
}
```

L'extrait de code Java 3 présente les mises en oeuvre des opérations `select`, `reject` et `collect` telles que définies dans la classe abstraite `AbstractSet<T>`. Ces méthodes sont définies dans la classe `AbstractSet`, et non dans `AbstractCollection`, parce que le type de la collection résultante est spécifique aux ensembles — `select` et `reject` retournent évidemment un sous-ensemble de l'ensemble de départ, alors que `collect` retourne plutôt un `Bag` (multi-ensemble).

Extrait de code Java 4 Exemple illustrant l'opération `collect` sur un `Set`.

```
Expression<Integer,Boolean> divisiblePar2 = new Expression<Integer,Boolean>() {
   public Boolean eval( Integer x ) {
        return x % 2 == 0;
   }
};

@Test public void collect() {
   Set<Integer> s = mkSet( 10, 20, 30, 40 );
   Bag<Boolean> b = mkBag( true, true, true, true );

   assertEquals( b, s.collect( divisiblePar2 ) );
}
```

La méthode de test présentée dans l'extrait de code Java 4 illustre ce comportement.

6.4 Mise en oeuvre de `iterate`

En OCL, l'opération `iterate` est la forme la plus générale d'opération s'appliquant à une collection, puisque de nombreuses autres opérations peuvent s'exprimer à l'aide d'`iterate`. En voici deux exemples :

1. ```
   Set{1,2,3,4}->iterate( x; tot: Integer = 0 | x + tot )
   =
   Set{1,2,3,4}->sum()
   =
   10
   ```

2. ```
   Set{1,2,3,4}->iterate( x; r: Set(Integer) = Set{} |
                          if x.mod(2) = 0 then r->including(x) else r endif )
   =
   Set{1,2,3,4}->select( x | x.mod(2) = 0 )
   =
   Set{2,4}
   ```

Comme on le constate, une opération `iterate` utilise une «expression» nécessitant deux arguments : un premier argument représentant l'élément courant de la collection, un deuxième argument représentant la valeur courante du résultat — appelé aussi l'*accumulateur*. Toutefois, on remarque aussi que la valeur initiale de cet accumulateur doit pouvoir être spécifiée explicitement, et pas nécessairement avec une valeur nulle ou vide :

3. ```
   Set{1,2,3,4}->iterate( x; prod: Integer = 10 | x * prod )
   =
   240
   ```

Le fichier Java 16 présente l'interface `BinaryExpression` requise pour la méthode `iterate`, méthode ayant alors la signature suivante (dans la classe abstraite `AbstractCollection<T>`) :

```
<R> R iterate( R initVal, BinaryExpression<T,R> expr );
```

Fichier Java 16 Spécification de l'interface `BinaryExpression`.

```
package OclCollections;

/**
 * Interface pour definir une notion d'expression binaire, utilisable
 * dans la methode iterate.
 */

public interface BinaryExpression<T,R> {
    public R eval( T v, R r );
}
```

L'extrait de code Java 5 présente ensuite la mises en oeuvre de l'opérations `iterate` telle que définie dans `AbstractCollection<T>` — rappelons que cette méthode est définie dans cette classe parce qu'applicable à n'importe quelle collection (ensemble, sac, séquence), et ce toujours avec la signature indiquée précédemment.

Finalement, la méthode de test présentée dans l'extrait de code Java 6 illustre l'utilisation de cette interface et de cette méthode.
```

Extrait de code Java 5 Mise en oeuvre de l'opération `iterate` dans la classe abstraite `AbstractCollection<T>`.

```
public <R> R iterate( R initVal, BinaryExpression<T,R> expr ) {
  R res = initVal;
  for ( T e : this ) {
    res = expr.eval( e, res );
  }
  return res;
}
```

Extrait de code Java 6 Exemples illustrant l'opération `iterate` sur des ensembles.

```
BinaryExpression<Integer,Integer> plus = new BinaryExpression<Integer,Integer>() {
    public Integer eval( Integer v1, Integer v2 ) {
      return v1 + v2;
    }
};

BinaryExpression<Integer,Integer> fois = new BinaryExpression<Integer,Integer>() {
    public Integer eval( Integer v1, Integer v2 ) {
      return v1 * v2;
    }
};

...
Set<Integer> s_1234 = mkSet(1,2,3,4);
...

@Test public void iterate_1() {
  Integer res = s_1234.iterate( 0, plus );
  assertEquals( (Integer) 10, res );
}

@Test public void iterate_2() {
  Set<Integer> empty = Collections.emptySet();
  Set<Integer> res =
       s_1234.iterate(
                 empty,
                 new BinaryExpression<Integer,Set<Integer>>() {
                     public Set<Integer> eval( Integer x, Set<Integer> r ) {
                       return x % 2 == 0 ? r.including(x) : r;
                 } } );
  assertEquals( mkSet(2,4), res );
}

@Test public void iterate_3() {
  Integer res = s_1234.iterate( 10, fois );
  assertEquals( (Integer) 240, res );
}
```

7 En guise de conclusion : Comparaison entre les collections `OclCollections` et les collections de la bibliothèque `java.util`

7.1 Collections de la bibliothèque `OclCollections`

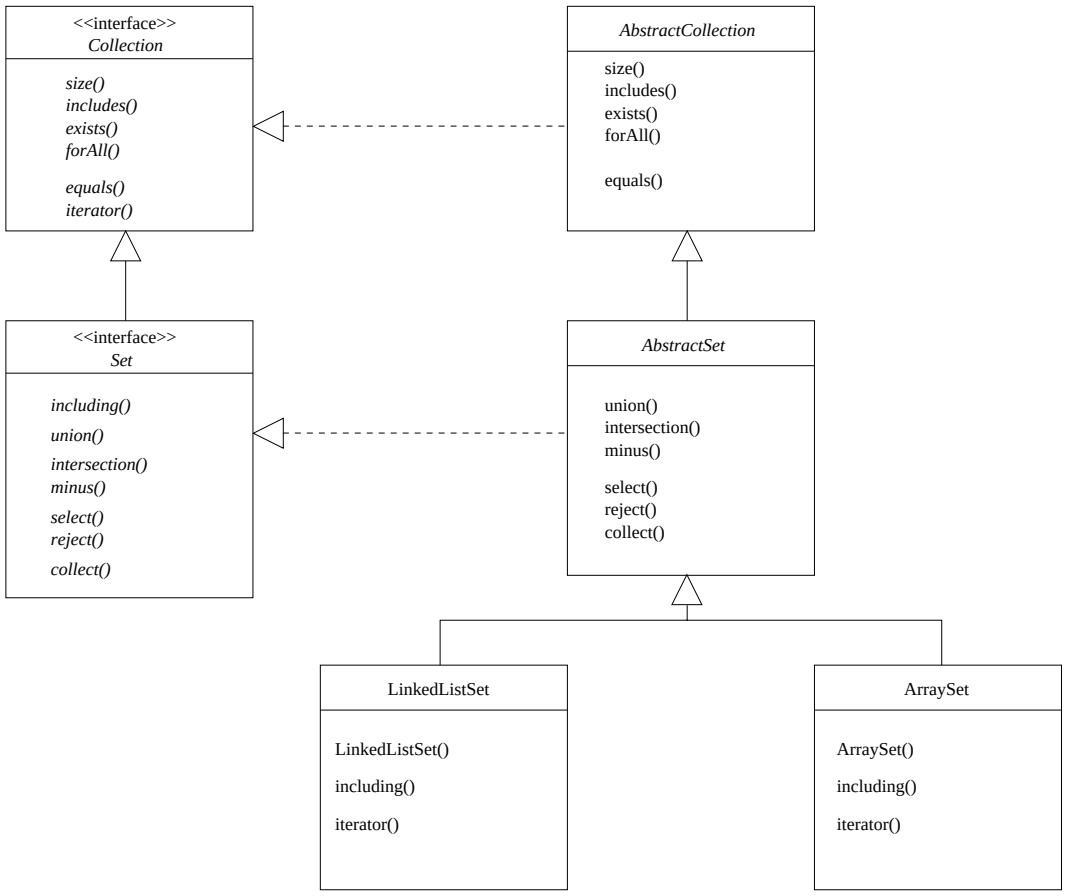

Figure 1: Relations entre les classes pour les ensembles de la bibliothèque `OclCollections`.

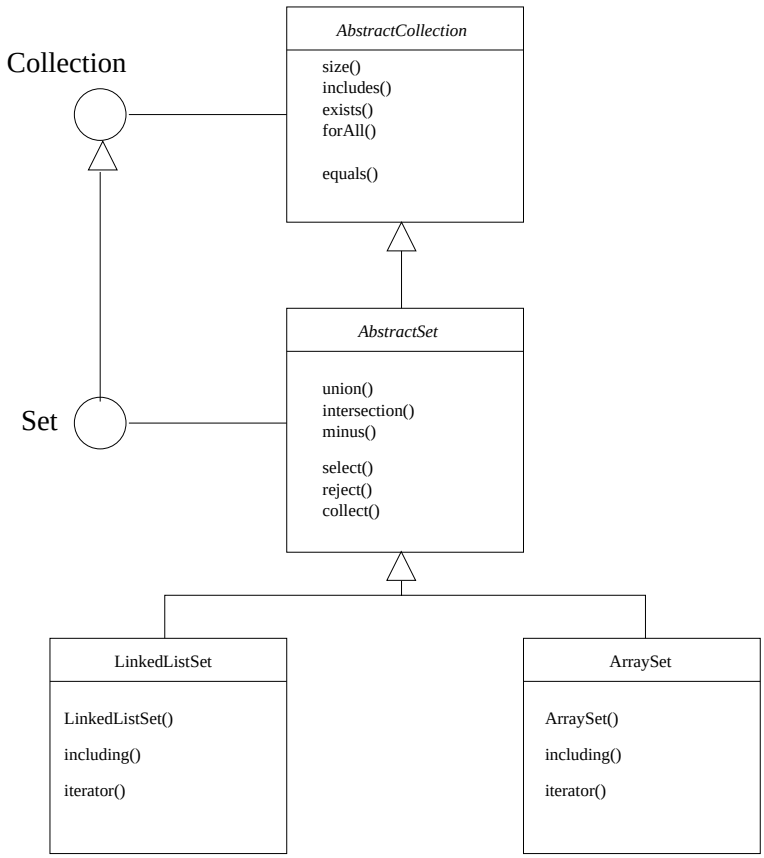

Figure 2: Relations entre les classes pour les ensembles d'`OclCollections` (bis).

La figure 1 illustre les relations entre les interfaces `Collection` et `Set`, les classes abstraites `AbstractCollection` et `AbstractSet` et les classes concrètes `LinkedListSet` et `ArraySet` — rappelons qu'un identificateur *en italiques* dénote une classe ou une méthode abstraite. La figure 2 illustre ces mêmes relations, mais cette fois en utilisant le stéréotype graphique pour une relation entre une interface et une classe qui met en oeuvre cette interface (diagramme avec «*lollipops*», en français, «suçons»).

Des relations semblables existent aussi entre, d'une part, `Sequence`, `AbstractSequence` et `LinkedListSequence`, d'autre part, `Bag`, `AbstractBag` et `LinkedListBag`, les autres types définis par la bibliothèque `OclCollections`. Soulignons toutefois que bien qu'il aurait été possible de développer une mise en oeuvre alternative en termes de tableaux pour les `Sequence` et `Bag`, celles-ci ne sont pas disponibles dans le code disponible sur le site Web du cours.

7.2 Collections de la bibliothèques `java.util`

La hiérarchie de classes pour les ensembles de la bibliothèque de collections Java est très semblable à celle de la bibliothèque `OclCollections`, telle qu'illustrée dans la figure 3.

Signalons toutefois certaines différences :

- L'ordre des éléments d'un `Set` (Fichier Java 17) est, comme pour les ensembles OCL, non significatif. Par contre, pour certains problèmes, il peut être utile d'énumérer les éléments de l'ensemble dans un ordre particulier. Un tel ensemble est alors représenté par un `SortedSet`. Bien qu'un type équivalent existe en OCL (`OrderedSet`), sa mise en oeuvre n'a pas été incluse dans la bibliothèque `OclCollections`.

- Les séquences sont représentées par un type `List` (Fichier Java 21).
- La notion de `Bag` n'est pas définie dans la bibliothèque Java. Par contre, on y trouve une notion de `Map`, appelée aussi dictionnaire (Fichier Java 22).

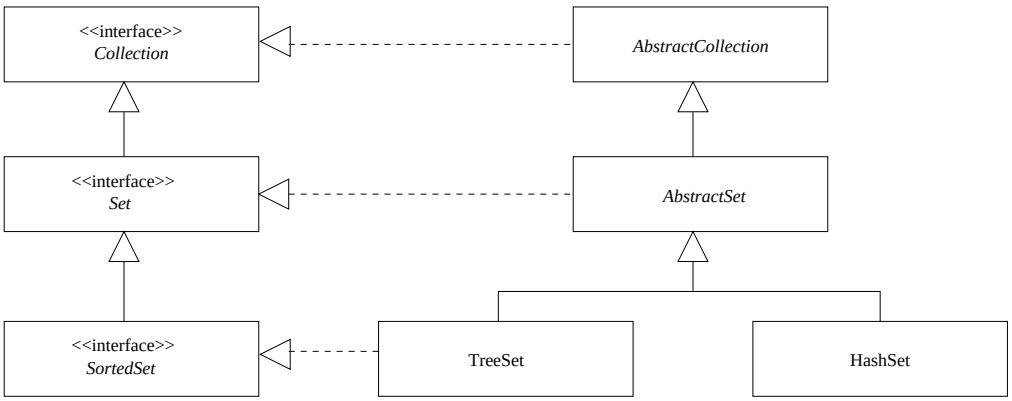

Figure 3: Relations entre les classes pour les ensembles de la bibliothèque `java.util`.

Fichier Java 17 Interface (partielle) pour les ensembles génériques de la bibliothèque des collections Java : `Set.java`.

```
interface Set<E> extends Collection<E> {

  boolean add( E o );
  // Adds the specified element to this set if it is not already present.

  boolean contains( Object o );
  // Returns true if this set contains the specified element.

  int size();
  // Returns the number of elements in this set (its cardinality).

  boolean addAll( Collection<? extends E> c );
  //Adds all of the elements in the specified collection to this set
  // if they're not already present.

}
```

Fichier Java 18 Classe de test illustrant les différences de comportement entre les ensembles OCL et les ensembles Java : `TestSetOCLJava.java`.

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.Iterator;

public class TestSetOCLJava {

    @Test public void including_vs_add() {
        // Cas OCL.
       OclCollections.Set<Integer> ss      = new OclCollections.LinkedListSet<Integer>();
       OclCollections.Set<Integer> ssAlias = ss;    // Creation d'un <<alias>>.

       ss.including( 10 );
       assertEquals( 0, ss.size() );                // Aucun effet de bord.
       assertFalse( ss.includes( 10 ) );
       assertEquals( 0, ssAlias.size() );           // Aucun effet sur l'alias.

        // Cas Java.
       java.util.Set<Integer> sj      = new java.util.HashSet<Integer>();
       java.util.Set<Integer> sjAlias = sj;         // Creation d'un alias.
       sj.add( 10 );
       assertEquals( 1, sj.size() );
       assertTrue( sj.contains( 10 ) );
       assertEquals( 1, sjAlias.size() );           // Effets de bord sur l'alias.
       assertTrue( sjAlias.contains( 10 ) );
    }

    @Test public void union_vs_addAll() {
        // Cas OCL.
       OclCollections.Set<Integer> ss1, ss2, ss3;
       ss1 = new OclCollections.LinkedListSet<Integer>().including( 10 ).including( 20 );
       ss2 = new OclCollections.LinkedListSet<Integer>().including( 30 ).including( 40 ).including( 50 );
       ss3 = new OclCollections.LinkedListSet<Integer>();

       ss1.union( ss2 );

       assertEquals( 2, ss1.size() );                 // Aucun effet de bord.
       assertEquals( 3, ss2.size() );
       assertEquals( 5, ss1.union( ss2 ).size() );

        // Cas Java.
       java.util.Set<Integer> sj1, sj1Alias, sj2;
       sj1 = sj1Alias = new java.util.HashSet<Integer>();  // Creation d'un alias.
       sj1.add( 10 );
       sj1.add( 20 );

       sj2 = new java.util.HashSet<Integer>();
       sj2.add( 30 );
       sj2.add( 40 );
       sj2.add( 50 );

       sj1.addAll( sj2 );

       assertEquals( 5, sj1.size() );
       assertEquals( 5, sj1Alias.size() );            // Effet de bord.
       assertEquals( 3, sj2.size() );
    }

}
```

Une différence plus fondamentale entre les collections de la bibliothèque `OclCollections` et celles de la bibliothèque `java.util` est la suivante : alors que la bibliothèque `OclCollections`

définit des «objets immuables» — appelés aussi «objets valeurs»> —, la bibliothèque `java.util` définit de véritables objets, donc des «entités mutables» (avec un état interne modifiable, qui évolue dans le temps).

Le fichier définissant *une partie* de l'interface pour les ensembles `java.util.Set` est présenté dans le Fichier Java 17 (interface `Set<E>`) — voir l'appendice E pour les autres classes de la bibliothèque (`List` et `Map`). On y remarque, entre autres, que l'appel à l'opération `add`, qui ajoute un élément dans l'ensemble, retourne aussi un `boolean` — qui indique si l'élément était déjà présent, ou non, dans l'ensemble —, contrairement à l'opération `including` de `OclCollections.Set` qui retourne un nouvel ensemble. L'opération `add( o )` de `java.util.Set` a donc *un effet de bord*, qui consiste à ajouter l'élément `o` à ceux déjà présents.

Le Fichier Java 18 (classe `TestSetOCLJava`) illustre, à l'aide d'un certain nombre de méthodes de test, cette différence de comportement entre les ensembles des deux bibliothèques de collections. Plus spécifiquement, on remarque que dans le cas d'OCL, où les ensembles sont des «valeurs», la présence d'alias[10] ne crée aucun danger d'effet de bord, puisque les opérations n'en ont pas. Par contre, pour les ensembles de la bibliothèque Java, qui sont des «objets», la présence d'alias entraine la possibilité d'effets de bord indirects : dans le premier cas (resp. le deuxième cas), `sjAlias` (resp. `sj1Alias`) est modifié *sans qu'on ait fait d'appel direct utilisant cet identificateur*!

A Interface, classe abstraite, classe concrète

Le tableau 1 est une traduction et adaptation de tableaux présentés par Haggar [Hag00, p. 62–63] pour décrire de façon synthétique les différences et similitudes entre interface, classe abstraite et classe concrète — rappelons qu'une classe est abstraite si elle possède au moins une méthode abstraite, i.e., dont la mise en oeuvre n'est pas spécifiée et doit l'être par une sous-classe.

[10] La présence d'un alias — on dit aussi synonyme — survient lorsqu'un identificateur réfère au même objet qu'un autre identificateur : l'un est donc un alias (un synonyme) pour l'autre.

Description	interface	classe abstraite	classe concrète
Description	Exprime un contrat sans mise en oeuvre	Exprime un contrat avec mise en oeuvre partielle	Mise en oeuvre concrète d'une interface ou d'une classe abstraite
À utiliser lorsque	On veut identifier un type abstrait, indépendamment de sa mise en oeuvre	On veut fournir une mise en oeuvre partielle de certaines opérations	On veut définir une mise en oeuvre concrète et détaillée
Contenu possible	Méthodes d'instance publiques et constantes statiques publiques seulement (pas de constructeurs)	Aucune contrainte	Aucune contrainte
Mise en oeuvre	Aucune mise en oeuvre permise	Mise en oeuvre partielle permise	Mise en oeuvre complète requise
Création d'instances	Non	Non	Oui
Héritage multiple	Oui	Non	Non

Tableau 1: Caractéristiques des interfaces, classes abstraites et classes concrètes

Rappelons, et c'est le cas avec les ensembles à la OCL que nous présentons, qu'une interface peut être mise en oeuvre par plusieurs classes concrètes différentes. De plus, comme on l'a aussi vu précédemment, une méthode de fabrication permet aussi de limiter l'espace du programme où les détails liés à la classe concrète sont spécifiés. Ainsi, il suffit d'indiquer la mise en oeuvre désirée à l'aide d'un appel approprié à la méthode `defineSetImplementation`, puis tous les appels pour créer un ensemble avec `emptySet()` ou `mkSet()` peuvent se faire sans être liés à une mise en oeuvre spécifique, donc sans faire appel au constructeur.

Finalement, soulignons qu'une classe abstraite (comme `AbstractSet`) qui fournit une mise en oeuvre de certaines méthodes en fonction d'autres méthodes définies dans les sous-classes représente une forme d'utilisation du patron de conception «*template method*» [GHJV95].

B Éléments additionnels sur JUnit 4.0

- Comme on l'a vu, l'annotation `@Before` permet d'assurer qu'une certaine méthode sera éxécutée *avant* l'exécution de chaque méthode de test — une telle méthode, dans les versions antérieures de JUnit, devait nécessairement s'appeler `setUp`. Dans nos exemples, une seule méthode était annotée de cette façon, mais il est possible d'annoter ainsi plusieurs méthodes.

- L'annotation `@After` permet d'assurer qu'une certaine méthode sera éxécutée *après* l'exécution de chaque méthode de test — méthode `tearDown` dans les anciennes versions de JUnit. On se sert d'une telle méthode pour effectuer, si nécessaire, un «nettoyage» du programme et des structures de données, par exemple, fermeture des fichiers.

- L'annotation `@BeforeClass`, resp. `@AfterClass`, devant une méthode assure que cette méthode sera exécutée avant, resp. après, l'exécution de la *suite de test* — donc avant, resp. après, l'exécution de l'ensemble des méthodes de test.

- Il est possible d'indiquer que l'exécution d'une méthode de test doit nécessairement générer une exception, par exemple :

```
@Test(expected=ArithmeticDivision.class)
  public void divisionPar0() {
    int n = 2 / 0;
  }
```

- L'annotation `@Ignore` permet de ne pas exécuter une méthode de test, mais sans devoir supprimer la méthode de test de la classe.Lors de l'exécution de la classe de test, un «I» sera alors affiché.

- Il est possible d'imposer une limite sur le temps d'exécution d'une opération, auquel cas le test échoue si la méthode de test ne s'exécute pas dans le délai prescrit. Dans l'exemple qui suit, le délai est de 100 ms :

```
@Test(timeout=100)
  public void testerTempsOp1() {
    executerOp1( ... );
  }
```

- Les différentes formes d'assertions sont les suivantes :

`assertTrue( boolean condition )`	Condition vraie
`assertFalse( boolean condition )`	Condition fausse
`assertNotNull( Object object )`	Référence non nulle
`assertSame( Object expected, Object actual )`	Références identiques
`assertNotSame( Object unexpected, Object actual )`	Références distinctes
`assertEquals( Object expected, Object actual )` `assertEquals( Object[] expecteds, Object[] actuals )`	Objets égaux Tableaux d'objets égaux
`assertEquals( double expected, double actual, double epsilon )`	Réels presqu'égaux
`fail()`	Échoue toujours

Pour chacune de ces méthodes, il existe aussi une *variante* avec un premier argument additionnel, de signature `String message`, qui permet, comme le deuxième argument d'un `assert` normal, d'indiquer un message d'erreur plus informatif.

C Règles pour l'écriture de classes et méthodes de test

Suggestions de Hunt et Thomas

Selon Hunt et Thomas [HT03], de bons tests doivent satisfaire, entre autres, les propriétés suivantes :

- Automatiques : L'ensemble des tests unitaires doit pouvoir s'exécuter de façon automatique, et ce à deux niveaux : l'invocation (exécution) des tests doit pouvoir se faire facilement, de même que la vérification que les bons résultats ont été produits. Ces deux propriétés sont supportées par un cadre de test tel que JUnit en utilisant correctement des assertions pour spécifier les comportements attendus.

- Complets : Les tests doivent couvrir «tout ce qui peut mal fonctionner». La notion de couverture des tests sera abordée plus en détail dans le cours INF3135 «Construction et maintenance de logiciels».

- Répétables : Si le logiciel n'a pas été modifié, l'exécution des tests devrait toujours produire le même résultat. Pour cela, il est important que les cas méthodes test puissent être exécutées *dans n'importe quel ordre*, sans que cela n'affecte les résultats.

- Indépendants : Chaque méthode de test doit être *cohésive* (donc bien ciblée) et indépendante des autres méthodes de test : « *When writing tests, make sure that you are only testing one thing at a time.* » [HT03, p. 83].

  Certains testeurs poussent cette idée de «cohésion» aussi loin que d'affirmer que chaque méthode de test *ne devrait contenir qu'une seule assertion*! Par contre, la plupart des testeurs ne vont pas aussi loin et permettent qu'une méthode de test contienne plusieurs assertions. Par contre, ces diverses assertions devraient, dans la mesure du possible, avoir un objectif commun, tester une fonctionalité bien spécifique.

  Quant à la notion d'indépendance des tests, rappellons qu'elle signifie aussi qu'une méthode de test ne devrait jamais dépendre, pour son bon fonctionnement, de l'exécution préalable d'une autre méthode de test.

Suggestions de Meszaros

Rappelons que, tel qu'indiqué plus haut (p. 6), Meszaros [Mes07] suggère que chaque test soit structuré sous la forme *Setup–Exercise–Verify–Teardown*. Entre autres, Meszaros indique ce qui suit [Mes07, p. 46] :

> *We avoid having a series of such alternating calls (setup, verify, exercise, teardown) because that approach would be trying to verify several distinct conditions—something that is better handled via distinct* Test methods.

Quant à la façon d'organiser et regrouper les différentes méthodes de tests, Meszaros suggère de débuter en utilisant une classe de test pour chaque classe à tester. Par contre, si (lorsque?!) le nombre de méthodes de test devient trop grand, il suggère alors de décomposer cette classe initiale de test en plusieurs classes, chaque classe de test pouvant alors être consacrée au test d'une «*feature*» — «*Although a "feature" of a class is typically a single operation or function, it may also be a set of related methods that operate on the same instance variable of the object.*» [Mes07, p. 625].

Finalement, concernant les noms des méthodes de test, Meszaros suggère d'utiliser des noms qui donnent une description complète du test, donc qui indique la méthode ou la *feature* qui est testée, les caractéristiques des arguments ou de l'état de départ (si approprié), et possiblement le résultat (ou l'état) attendu. Par exemple [Mes07, chap. 24], supposons qu'on doive tester une classe `Flight` qui possède trois états possibles — `Unscheduled`, `Scheduled` et `AwaitingApproval` — et quatre méthodes — `schedule`, `requestApproval`, `deSchedule` et `approve`. Les noms des tests auraient alors l'allure suivante — en utilisant la suggestion de Ford (ci-bas) :

```
request_approval_from_scheduled_state
request_approval_from_unscheduled_state
request_approval_from_awaiting_approval_state
schedule_from_unscheduled_state
schedule_from_scheduled_state
...
```

Suggestions de Meyer

Bertrand Meyer suggère sept principes liés aux tests, les quatre premiers étant les suivants [Mey08] :

- **Principe 1. Définition** Tester un programme consiste à essayer de le faire échouer (de le faire «planter»).
- **Principe 2. Tests versus specs** Les tests ne remplacent pas les spécifications.
- **Principe 3. Test de régression** Toute exécution qui conduit à une erreur devrait générer une méthode de test, qui deviendra de façon permanente un élément de la suite de test.
- **Principe 4. Utilisation d'oracles** Déterminer le succès ou l'échec d'un test doit se faire de façon automatique.

Meyer mentionne aussi ce qui suit (entrevue accordée en 2003) :[11]

> *[T]here's no such thing as an exhaustive test, a test that exercises all possible cases. So we know we can't have an exhaustive test, but we can have systematic tests that have a likelihood of exercising the cases that will fail. For example, if you have a parameter that must be between certain bounds, then you want to test the values close to the bounds of the range. You want to test maybe the value in the middle, and maybe a few in-between.*

Suggestion de Ford

Dans son livre «*The Productive Programmer*» [For08], Ford suggère, parce que les noms de test doivent généralement être longs et descriptifs, de ne pas utiliser, *même en Java*, le style `camelCase` pour les noms de test. Donc, plutôt que d'écrire un nom de test tel que `facteursPourNombresDivisiblesPar3`, il suggère plutôt d'écrire `facteurs_pour_nombres_divisibles_par_3` ce qui est plus lisible.

Suggestions de Osherove

Dans son blog[12], Roy Osherove suggère ce qui suit pour les noms de test :

- *Test name should expres a specific requirement.*
- *Test name should include the expected input or state and the expected result for that input or state.*
- *Test name should include name of tested method or class.*

D *Put Angels on Your Shoulders*

Ce qui suit est une série d'extraits (ma traduction) du chapitre «*Put Angels on Your Shoulders*» du livre «*Practices of an Agile Developer: Working in the Real World*» de Subramaniam et Hunt [SH05], chapitre qui traite des avantages et bénéfices des tests unitaires :

- **Unit testing provides instant feedback.** Votre code est testé et exécuté de façon répétitive. Au fur et à mesure où vous modifiez et réécrivez votre code, les cas de test permettent de vérifier que vous n'avez pas brisé les contrats existants. Vous pouvez donc identifier et corriger rapidement les problèmes.
- **Unit testing makes your code robust.** Tester un module aide à réfléchir à son comportement, à penser aux différents cas possibles, positifs, négatifs et exceptionnels.

[11] `http://www.artima.com/intv/contest.html`
[12] `http://weblogs.asp.net/rosherove/archive/2005/04/03/TestNamingStandards.aspx`

- **Unit testing can be a helpful design tool.** Le développement de tests unitaires aide souvent à produire une solution simple et élégante.
- **Unit testing is a confidence booster.** Vous avez testé votre code et vérifié son comportement dans une grande variété de conditions : ceci vous donnera confiance lorsque vous aurez à faire face à de nouvelles tâches, avec des délais serrés.
- **Unit tests can act as probes when solving problems.** Les tests jouent un rôle semblable à celui d'un oscilloscope pour un circuit, aidant à mieux comprendre son comportement interne. Ceci aide à mieux cerner la cause d'un problème.
- **Unit tests are reliable documentation.** Lorsque vous devez apprendre une nouvelle API, les tests unitaires servent de documentation fiable et précise de cette API.
- **Unit tests are a learning aid.** Lorsque vous commencez à utiliser une nouvelle API, vous pouvez écrire des tests qui permettent de vérifier que vous comprenez correctement cette API.

En conclusion :

> **Use automated unit tests.** *Good unit tests warn you about problems immediately. Don't make any design or code changes without solid unit tests in place.*

E Les interfaces pour les collections de la bibliothèque `java.util`

Cette annexe présente, mais sans les expliquer, les principales interfaces — mais pas les classes abstraites ou concrètes associées — de la bibliothèque `java.util`. Les interfaces pour les ensembles et pour les séquences, soit `Set` et `List`, sont toutes deux des extensions de l'interface `Collection`, qui regroupe les éléments communs aux ensembles et séquences (ajout, taille, etc.). Quant aux dictionnaires, ils sont définis par l'interface `Map`. Signalons simplement que les méthodes qui ont une mise en oeuvre par défaut dans les classes abstraites correspondantes sont indiquées comme étant «`optional`» dans l'interface.

Pour plus de détails, voir le site Web de Sun, d'où sont tirées ces interfaces : `http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html`

Fichier Java 19 Interface pour les collections génériques de la bibliothèque des collections Java : `Collection.java`.

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Fichier Java 20 Interface pour les ensembles génériques de la bibliothèque des collections Java : `Set.java`.

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Fichier Java 21 Interface pour les séquences génériques de la bibliothèque des collections Java : `List.java`.

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

Fichier Java 22 Interface pour les dictionnaires génériques de la bibliothèque des collections Java : `Map.java`.

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Références

[Bec00] K. Beck. *Extreme Programming Explained—Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[Bec01] K. Beck. Aim, fire. *IEEE Software*, 18(6):87–89, 2001.

[Bec07] K. Beck. *Implementation Patterns*. Addison-Wesley, 2007.

[BG98] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[Fla02] D. Flanagan. *Java in a nutshell, Third Edition*. O'Reilly, 2002.

[For08] N. Ford. *The Productive Programmer*. O'Reilly, 2008.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Hag00] P. Haggar. *Practical Java—Programming Language Guide*. Addison-Wesley, 2000.

[HT00] A. Hunt and D. Thomas. *The Pragmatic Programmer—From journeyman to master*. Addison-Wesley, Reading, MA, 2000. [QA76.6H858].

[HT03] A. Hunt and D. Thomas. *Pragmatic Unit Testing In Java with JUnit*. The Pragmatic Bookshelf, Raleigh, NC, 2003.

[Mes07] G. Meszaros. *xUnit Test Patterns—Refactoring Test Code*. Addison-Wesley, Upper Saddle River, NJ, 2007.

[Mey08] B. Meyer. Seven principles of software testing. *IEEE Computer*, 41(8):99–101, Aug. 2008.

[RK03] P.N. Robillard and P. Kruchten. *Software Engineering Process with the UPEDU*. Addison-Wesley, 2003.

[SH05] V. Subramaniam and A. Hunt. *Practices of an Agile Developer: Working in the Real World*. The Pragmatic Bookshelf, 2005.