Conception et programmation par contrats en Java (iContract, JML et Modern JASS) et en Eiffel

Hiver 2010

Table des matières

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

Douglas Adams

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

Douglas Adams

1 Introduction

Dans ce chapitre, nous allons examiner des exemples de contrats formels spécifiés dans divers langages : iContract [Ens01], JML [LCC+03, BCC+05], Modern JASS [Rie07][1] et Eiffel [Mey97].

Le but de ce chapitre n'est pas de vous faire apprendre de nouveaux langages de spécification. En d'autres mots, l'objectif n'est pas qu'après avoir lu ce chapitre, vous soyez capables d'écrire des spécifications dans ces notations et langages. L'objectif de ce chapitre est plutôt de vous faire voir que maintenant que vous connaissez un langage de spécification formelle, à savoir OCL [WK03], et que vous comprenez l'idée de contrat formulé à l'aide de pré/postconditions, alors vous êtes assurément capable **de lire et de comprendre** des contrats exprimés dans divers autres langages — donc d'utiliser des modules ou composants logiciels qui seraient spécifiés à l'aide de tels contrats.

L'autre aspect important de ce chapitre est de voir que la notion de contrat ne sert pas uniquement à l'étape de spécification de logiciels, mais peut servir aussi à l'étape de construction de logiciels. Ceci signifie qu'on peut utiliser, de façon concrète et effective, les contrats *dans des programmes compilables et exécutables.* Ainsi, les exemples que nous allons voir sont des exemples **de programmes** écrits soit en Java — avec diverses bibliothèques ou extensions (préprocesseurs) — soit en Eiffel — le premier langage de programmation à avoir introduit la notion de contrat. Il s'agit donc de contrats exprimés dans des *langages de programmation* — et non, comme en OCL, dans un langage de spécification — contrats qui peuvent alors être *évalués,* **donc vérifiés***, au moment de l'exécution du programme.*

Contenu = Exemples de contrats exprimés dans divers langages :

- iContract (Java)
- JML (Java)
- Modern JASS (Java)
- Eiffel

Objectif = Vous faire voir que, connaissant OCL, vous pouvez lire et comprendre de tels contrats

[1] `http://modernjass.sourceforge.net`

2 Pourquoi évaluer dynamiquement les contrats

On reconnait de plus en plus l'importance que peuvent jouer la spécification de contrats et l'évaluation dynamique (durant l'exécution) de ces contrats. C'est ce qui explique que, de plus en plus, les langages de programmation incluent des constructions ou définissent des extensions qui permettent de spécifier et vérifier de tels contrats — par exemple, à l'aide d'une instruction `assert` simple, ou à l'aide de constructions plus sophistiquées, comme on le verra plus loin.

Ainsi, comme l'indique la documentation du langage Java[2] au sujet des assertions en général, les pré/postconditions pouvant être vues comme une forme particulière d'assertions :

> *Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs. As an added benefit, assertions serve to document the inner workings of your program, enhancing maintainability.*

La notion de contrat — et les contraintes associées en termes de pré/postconditions — peut donc être appliquée à l'étape de construction (codification et programmation, tests unitaires) pour aider au **débogage** de programmes. Ainsi, si durant l'exécution d'un programme il arrive qu'une assertion (précondition, postcondition, ou autre) ne soit pas vérifiée, alors c'est qu'il y a une *erreur* (un *bogue*) *dans le programme.*

Plus précisément, dans le cas des pré/postconditions :

- La violation d'une précondition est le signe d'une erreur dans le code du «*client*» : le client ne s'est pas assuré avant l'appel du service, comme il aurait dû le faire, que les arguments fournis étaient valides pour le service qu'il désirait utiliser. L'exécution du service demandé *ne devrait donc pas* se faire.
- La violation d'une postcondition est le signe d'une erreur dans le code du «*fournisseur*» : le fournisseur n'a pas été capable de fournir le service demandé. Donc, si la situation est telle qu'il est impossible pour le serveur d'établir et satisfaire la postcondition, alors l'exécution du programme devrait tout simplement être **avortée**.

3 Préconditions vs. exceptions

> *Use Assertions to Prevent the Impossible : Assertions validate your assumptions. Use them to protect your code from an uncertain world.*
>
> *Use Exceptions for Exceptional Problems : Exceptions can suffer from all the readability and maintainability problems of classic spaghetti code. Reserve exceptions for exceptional things.*
>
> *Hunt & Thomas, The Pragmatic Programmer [HT00].*
>
> Bref, pour résumer :
>
> - Assertion = test si connerie de toi
> - Exception = test si connerie des autres
>
> Zenitram (post sur un site Web) [3]

Dans le langage **de spécification** OCL, la notion d'exception n'existe pas. Les exemples vus en OCL utilisaient donc tous des préconditions pour assurer qu'une opération s'exécute

[2] `http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`
[3] `http://geekexplains.blogspot.com/2008/06/asserions-in-java-assertions-vs.html`

uniquement si les arguments fournis (ou les attributs ou propriétés associés à l'état du système ou de l'objet) sont valides

Par contre, la plupart des langages **de programmation** ont des **mécanismes d'exception**. On peut donc se poser la question de savoir quand on doit utiliser des assertions (préconditions) par opposition à quand on doit utiliser des exceptions.

Dans un langage de programmation supportant les deux mécanismes, le rôle des assertions et le rôle des exceptions peuvent être caractérisés comme suit :

- Le rôle des assertions — y compris les contrats spécifiés avec des pré/postconditions — est de vérifier *que les choses qui ne peuvent pas arriver ne se produisent pas.*

  Donc, si une assertion est fausse, c'est qu'il y a une erreur (un bogue) dans le programme. Donc, il faut avorter l'exécution.

- Le rôle des exceptions est de traiter/signaler certains *cas particuliers*, qui peuvent être associés à des erreurs, mais pas nécessairement.

En d'autres mots, comme le suggère la citation ci-haut :

- Si la condition dont on veut vérifier la validité est liée à la mise en oeuvre privée/interne du logiciel — Liskov et Guttag parlent de «contexte local d'utilisation» [LG01] —, alors c'est probablement une assertion qu'il faut utiliser.

- Si la condition dont on veut vérifier la validité est liée à des condition sur des arguments qui proviennent «de l'extérieur» (du monde réel, via des entrées/sorties, ou de modules–clients qui ne sont pas sous «notre» responsabilité — personnelle ou de notre équipe de développement), alors c'est probablement une exception qu'il faut utiliser.

Question : Quand doit-on utiliser des préconditions par opposition à des exceptions?

- Assertions : vérifier *que les choses qui ne peuvent pas arriver ne se produisent pas*

- Exceptions : signaler certains «*cas particuliers*»

Donc :

- Condition liée à la mise en oeuvre (interne) du logiciel : assertion
- Condition liée à des éléments qui proviennent «de l'extérieur» : exception

Documentation Java : «*Do not use assertions for argument checking in public methods.*»
Signalons que la documentation Java indique ce qui suit quant à l'utilisation des assertions — le *public methods* est important :[4]

- *Do not use assertions for argument checking in public methods.*

  Argument checking is typically part of the published specifications (or contract) of a method, and these specifications must be obeyed whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException). An assertion failure will not throw an appropriate exception.

- *Do not use assertions to do any work that your application requires for correct operation.*

  Because assertions may be disabled, programs must not assume that the boolean expression contained in an assertion will be evaluated. Violating this rule has dire consequences.

[4] `http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`

3.1 Exemple

Supposons qu'on veuille définir une fonction Java réalisant la méthode dont la spécification OCL est présentée à l'exemple 1.

Exemple 1 Spécification OCL d'une fonction (pure) pour calculer la racine carrée d'un nombre réel.

```
racineCarree( x: Real, precision: Real ): Real
 pre XNonNegatif:
   x >= 0.0
 pre PrecisionPositive:
   precision > 0.0

 post RacinePositive:
   result >= 0.0
 post BonneRacine:
   approxEgal( result*result, x, precision )

approxEgal( x1: Real, x2: Real, precision: Real ): Boolean
  = (x1 - x2).abs() <= precision
```

Deux alternatives sont possibles quant à une méthode concrète Java :

1. Il s'agit d'une fonction *non publique*[5], utilisée à l'interne, par nos propres modules ou composants ou par des modules ou composants étroitement liés (par exemple, même équipe de développement). Dans ce cas, on pourrait alors avoir une mise en oeuvre ayant l'allure suivante :

   ```
   static float racineCarree( float x, float precision ) {
     assert x >= 0.0 && precision > 0.0;
     ...
   }
   ```

2. Il s'agit d'une fonction publique, utilisée par divers «clients» sur lesquels nous n'avons «aucun contrôle», que nous ne connaissons pas à l'avance. Dans ce cas, il serait préférable d'avoir la possibilité de signaler une exception, laissant ainsi au client la possibilité de récupérer l'erreur et tenter de la corriger :

   ```
   public static float racineCarree( float x, float precision )
                                     throws RacineImaginaire, PrecisionInvalide {
     if ( x < 0.0 )          { throw new RacineImaginaire(...); }
     if ( precision <= 0.0 ) { throw new PrecisionInvalide(...); }
     ...
   }
   ```

   Comme il s'agit ici d'une fonction publique et utilisable par n'importe quel client, il est possible que l'erreur sur l'argument puisse être récupérée et corrigée. Par exemple, imaginons une interface personne–machine qui lit un nombre réel et utilise cette méthode publique pour en calculer la racine carrée. Si l'argument est invalide et qu'on signale l'exception, le module d'interface personne–machine pourrait alors indiquer à l'usager que l'argument n'était pas valide et qu'il doit donc fournir une autre valeur, appelant ensuite à nouveau la méthode publique.

[5] Rappelons qu'en Java, une méthode qui n'est pas publique n'est pas nécessairement privée : elle peut être `private`, `protected` ou, lorsque sans modificateur de visibilité, accessible au niveau classe et paquetage seulement.

On verra dans ce qui suit que, dans certaines notations pour la spécification de contrats, les deux formes peuvent être spécifiées à l'aide de contrats appropriés.

4 Exemples iContract

L'outil iContract est un *préprocesseur* pour programmes Java, préprocesseur qui analyse des commentaires spéciaux écrits dans un style semblable à `JavaDoc` et qui génère un programme Java équivalent, mais où des assertions appropriées ont été ajoutées de façon à vérifier dynamiquement les pré/postconditions ainsi que les invariants.

Les deux exemples qui suivent sont tirés de «*Design by Contract, by Example*» [MM02, Chap. 11].

4.1 Spécification pour une classe `Queue`

Le premier exemple iContract (Exemple 2) est celui d'une spécification pour une classe `Queue`, i.e., une file FIFO (*First-In, First-Out*).[6]

Quelques caractéristiques de cette spécification fondée sur les contrats avec pré/postconditions, et qui utilise un style DBC tel que décrit dans le chapitre sur les types abstraits (notes de cours sur OCL) :

- Il n'y a pas de modèle spécifié explicitement pour décrire l'état de l'objet. L'état de l'objet est plutôt décrit par les méthodes de type *observateur* associées à l'objet.

  Ces méthodes de type *observateur* sont donc prises comme *primitives* — des désignations, non spécifiées de façon détaillée et explicite. Le comportement des autres méthodes (constructeurs, mutateurs ou observateurs dérivés) est alors décrit en spécifiant l'effet de la méthode sur ces observateurs primitifs.

- Les préconditions sont indiquées par `require` (nécessite) alors que les postconditions sont indiquées par `ensure` (assure).

- L'identificateur `return` dénote le résultat retourné par une méthode (lorsque son type n'est pas `void`).

- Pour référer à l'état avant un changement d'état, on utilise `@pre` en suffixe, comme en OCL.

4.2 Spécification pour une interface `IDictionary`

Le deuxième exemple iContract (Exemple 3) est celui d'une spécification pour une interface `IDictionary`, i.e., un dictionnaire.[7]

5 Exemples JML

Depuis quelques années, l'outil JML est devenu très populaire dans le monde des spécifications formelles. Tout comme l'outil iContract, JML est un *préprocesseur* pour programmes Java annotés de commentaires spéciaux à la `JavaDoc`. Un programme Java annoté de spécifications JML doit tout d'abord être compilé par un compilateur spécial, appelé `jmlc`. Le fichier `.class` résultant peut ensuite être exécuté par une machine Java normal. Par contre, pour

[6] L'exemple a été légèrement modifié par rapport à l'exemple original [MM02] pour en faire une classe générique.

[7] L'exemple a été légèrement modifié par rapport à l'original [MM02] pour en faire une interface générique.

Exemple 2 Spécification d'une classe `Queue` avec iContract (adaptée de [MM02]).

```
public class Queue<T> {
    /**
     * Construct a new, empty, queue.
     *
     * @ensure this.size() == 0
     */
    public Queue();

    /**
     * Return the number of elements in the receiver.
     */
    public int size() { ... }

    /**
     * Return the element at position i of the receiver (first element is at position 0).
     *
     * @require i >= 0 && i <= this.size()-1
     */
    public T get( int i ) { ... }

    /**
     * Return the head element of the receiver.
     *.
     * @require this.size() >= 1
     *
     * @ensure return == this.get(0)
     */
    public T head() { ... }

    /**
     * Return true if, and only if, the receiver contains no elements.
      *
     * @ensure return == (this.size() == 0)
     */
    public boolean isEmpty() { ... }

    /**
     * Return a shallow copy of the receiver.
     *
     * @ensure return.size() == this.size()
     * @ensure forall int i in 0..this.size()-1 |
     *       this.get(i) == return.get(i)
     */
    public Queue<T> shallowCopy() { ... }

    /**
     * Add the element e to the end of the receiver.
     *
     * @require e != null
     *
     * @ensure this.size() == this.size()@pre + 1
     * @ensure this.get( this.size()-1 ) == e
     */
    public void put( T e ) { ... }

    /**
     * Remove the head-element from the receiver.
     *
     * @require this.size() >= 1
     *
     * @ensure this.size() == this.size()@pre - 1
     * @ensure forall int i in 0..this.size()-2 |
     *          this.shallowCopy()@pre.get(i+1) == this.get(i)
     */
    public void remove() { ... }
}
```

Exemple 3 Spécification d'une interface `IDictionary` avec iContract (adaptée de [MM02]).

```
/**
 * @invariant size() >= 0
 */
public interface IDictionary<K,V> {
    public int size();


    /**
     * @require key != null
     * @ensure (size() == 0) implies !return
     */
    boolean has( K key );


    /**
     * @require key != null
     * @require has(key)
     */
    V valueFor( K key );

    /**
     * @ensure return == (size() == 0)
     */
    boolean isEmpty();


    /*
     * @require key != null
     * @require !has(key)
     *
     * @ensure size() == size()@pre + 1
     * @ensure has(key)
     * @ensure valueFor(key) == value
     */
    void put( K key, V value );


    /*
     * @require key != null
     * @require has(key)
     *
     * @ensure size() == size()@pre - 1
     * @ensure !has(key)
     */
    void remove( K key );
}
```

une vérification plus complète des pré/postconditions et assertions, on peut aussi utiliser `jmlrac` (JML *run-time assertion checker*).

Une particularité de JML, et c'est ce qui fait son intérêt, est que de nombreux outils additionnels ont été développés, entre autres : un outil pour vérifier *à la compilation* diverses propriétés dans le but de prévenir les erreurs d'exécution ; un outil pour générer automatiquement des tests unitaires JUnit à partir des spécifications ; un outil pour générer de la documentation sous une forme plus lisible.

5.1 Description informelle du type `JMLEqualsSet`

Comme la plupart des langages de spécification formelle, JML possède une riche bibliothèque de types pouvant être utilisés dans les spécifications : `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs`

Comme en OCL, ces types définissent *des collections de valeurs*, et non des classes d'objets.

La figure 1[8] présente des extraits de la description du type `JMLEqualsSet`, le type pour des ensembles *immuables* de la bibliothèque JML — le `Equals` indique que les éléments des ensembles doivent pouvoir être comparés entre eux avec `equals`. Signalons, dans cete description des méthodes, l'utilisation fréquente de `Object` : JML, dans sa version actuelle, n'a pas encore intégré les types génériques de Java 5 `:(`

5.2 Une fonction pour calculer la racine carrée d'un nombre

Le premier exemple (Exemple 4) est une spécification simple pour une fonction qui calcule la racine carrée d'un nombre réel, semblable à celle vue en OCL (chapitre sur les fonctions) à la différence que le contrat indique qu'une exception doit être signalée si le nombre reçu en argument est négatif.

Exemple 4 Spécification JML d'une fonction racine carrée.

```
import org.jmlspecs.models.JMLDouble;

public class Sqrt {
    public final static double precision = 0.001;

    /*@ public normal_behavior
      @   requires x >= 0.0;
      @   ensures JMLDouble.approximatelyEqualTo( x, \result * \result, precision );
      @ also
      @ public exceptional_behavior
      @   requires x < 0.0;
      @   signals (Exception e) e instanceof RacineImaginaireException;
      @*/
    public static double sqrt( double x ) {
       return Math.sqrt( x );
    }
}
```

5.3 Une spécification JML pour une interface `Stack`

Le deuxième exemple JML (Exemple 5), tiré directement de l'article original [LCC+03], présente une spécification pour une interface `Stack` (pile non bornée).

[8] `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/org/jmlspecs/models/JMLEqualsSet.html`

```
==================
Class JMLEqualsSet
==================

Constructor Summary
-------------------

JMLEqualsSet()
          Initialize this to be an empty set.

JMLEqualsSet(Object e)
          Initialize this to be a singleton set containing the given element.

Method Summary
--------------

Object choose()
          Return an arbitrary element of this.

boolean containsAll(non_null Collection c)
          Tell whether, for each element in the given collection, there is a ".equals" element in this set.

JMLEqualsSet difference(non_null JMLEqualsSet s2)
          Returns a new set that contains all the elements that are in this but not in the given argument.

JMLEqualsSetEnumerator elements()
          Returns an Enumeration over this set.

boolean has(Object elem)
          Is the argument ".equals" to one of the objects in theSet.

JMLEqualsSet insert(Object elem)
          Returns a new set that contains all the elements of this and also the given argument.

int int_size()
          Tells the number of elements in the set.

JMLEqualsSet intersection(non_null JMLEqualsSet s2)
          Returns a new set that contains all the elements that are in both this and the given argument.

boolean isEmpty()
          Is the set empty.

boolean isSubset(non_null JMLEqualsSet s2)
          Tells whether this set is a subset of or equal to the argument.

JMLEqualsSet remove(Object elem)
          Returns a new set that contains all the elements of this except for the given argument.

JMLEqualsSet union(non_null JMLEqualsSet s2)
          Returns a new set that contains all the elements that are in either this or the given argument.
```

Figure 1: Extraits de la spécification des ensembles `JMLEqualsSet` de la bibliothèque des types JML.

Exemple 5 Spécification JML pour une interface `Stack` (tiré de [LCC+03]).

```
//@ model import org.jmlspecs.models.*;
public interface Stack {
  //@ public model instance JMLObjectSequence absVal;

  //@ public instance invariant absVal != null;

  /*@ public normal_behavior
    @   requires true;
    @   assignable absVal;
    @   ensures absVal.equals(\old(absVal.insertFront(x))); @*/
  void push(Object x);

  /*@   public normal_behavior
    @     requires !absVal.isEmpty();
    @     assignable absVal;
    @     ensures absVal.equals(\old(absVal.trailer()))
    @          && \result == \old(absVal.first());
    @ also
    @   public exceptional_behavior
    @     requires absVal.isEmpty();
    @     assignable \nothing;
    @     signals (Exception e)
    @          e instanceof IllegalStateException; @*/
  Object pop();

  //@ ensures \result <==> absVal.isEmpty();
  /*@ pure @*/ boolean isEmpty();
}
```

Fig. 1. The specification and code for the interface `Stack`.

5.4 Une spécification JML pour une classe `Purse` modélisant un porte-monnaie électronique

Le troisième exemple JML (Exemple 6), tiré lui aussi directement de l'article original [BCC+05], présente une spécification pour un «porte-monnaie électronique» simple — classe `Purse`.

Quelques éléments nouveaux :

- Le type `JMLObjectSequence` est équivalent au type OCL `Sequence(Object)`, à la différence que les éléments Java sont comparés entre eux avec «==» et non avec `equals` — encore une fois, `Object` est utilisé pour émuler la généricité.

  Les opérations `first()` et `trailer()` sur de telles séquences retournent, respectivement, le premier élément de la séquence (équivalent de `s->at(1)`) et toute la séquence sauf le premier élément (équivalent de `s->subSequence(2, s->size())`).

- Une clause `assignable x;` indique que `x` est un champ du modèle qui sera modifié suite à l'exécution de la méthode.

6 Exemples Modern JASS

L'outil Modern JASS [Rie07][9], tout comme iContrat et JML, est une extension de Java. Toutefois, à la différence de ces deux langages, les contrats sont spécifiés à l'aide d'annotations — annotations introduites en Java 5.0, comme celles utilisées en JUnit 4.0 pour indiquer les méthodes de tests et autres méthodes spéciales.

Le traitement de ces annotations se fait donc à l'aide d'un processeur d'annotation, indiqué à la compilation et à l'exécution à l'aide d'un fichier `.jar`, compilation et exécution se faisant avec `javac` et `java`.

6.1 Une fonction pour calculer la racine carrée d'un nombre

Le premier exemple (Exemple 7) est pour une fonction qui calcule la racine carrée d'un nombre réel.

Quelques éléments à souligner :

- Les contrats et les mots clés utilisés dans les contrats sont indiqués à l'aide d'annotations — indiquées par le préfixe «`@`».

- L'argument d'une annotation pour une précondition ou une postcondition est une *chaîne de caractères*, chaîne qui doit représenter une expression *qui sera syntaxiquement et sémantiquement correcte lorsqu'évaluée dans le contexte du corps de la méthode.* Dans une telle expression, on peut donc utiliser les arguments de la méthode (ici, `x`) et, lorsque la méthode retourne un résultat (type différent de `void`), on peut utiliser `@Result` (donc comme on le fait en OCL avec `result`) dans l'expression pour la postcondition.

- Contrairement à la version JML dont le contrat spécifiait qu'une exception devait être signalée si le nombre reçu en argument était négatif, celui-ci spécifie un contrat vraiment semblable à celui de l'exemple vu en OCL (chapitre sur les fonctions) : aucun résultat particulier n'est spécifié dans le cas où le nombre est négatif.

  Si on exécute le petit programme de test décrit par la méthode `main` de l'exemple 7, on obtient alors le résultat d'exécution suivant :

[9] L'outil s'appelle «Modern JASS» car il s'agit en fait d'une révision et réimplémentation d'un outil précédent qui s'appelait «JASS» : Java ASSertion.

Exemple 6 Spécification JML pour une classe `Purse` modélisant un porte-monnaie électronique (tirée de [BCC+05]).

```
public class Purse {

    final int MAX_BALANCE;
    int balance;
    //@ invariant 0 <= balance && balance <= MAX_BALANCE;

    byte[] pin;
    /*@ invariant pin != null && pin.length == 4
      @           && (\forall int i; 0 <= i && i < 4;
      @                              0 <= pin[i] && pin[i] <= 9);
      @*/

    /*@ requires   amount >= 0;
      @ assignable balance;
      @ ensures    balance == \old(balance) - amount
      @            && \result == balance;
      @ signals (PurseException) balance == \old(balance);
      @*/
    int debit(int amount) throws PurseException {
        if (amount <= balance) { balance -= amount; return balance; }
        else { throw new PurseException("overdrawn by " + amount); }
    }

    /*@ requires   p != null && p.length >= 4;
      @ assignable \nothing;
      @ ensures    \result <==> (\forall int i; 0 <= i && i < 4;
      @                                         pin[i] == p[i]);
      @*/
    boolean checkPin(byte[] p) {
        boolean res = true;
        for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
        return res;
    }

    /*@ requires   0 < mb && 0 <= b && b <= mb
      @            && p != null && p.length == 4
      @            && (\forall int i; 0 <= i && i < 4;
      @                               0 <= p[i] && p[i] <= 9);
      @ assignable MAX_BALANCE, balance, pin;
      @ ensures    MAX_BALANCE == mb && balance == b
      @            && (\forall int i; 0 <= i && i < 4; p[i] == pin[i]);
      @*/
    Purse(int mb, int b, byte[] p) {
        MAX_BALANCE = mb; balance = b; pin = (byte[]) p.clone();
    }
}
```

Fig. 1. Example JML specification

Exemple 7 Spécification Modern JASS d'une fonction racine carrée.

```
package Exemples;

import jass.modern.*;

public class Sqrt {

    public final static double eps = 0.1;

    // Note: L'utilisation d'une fonction statique ne semble pas
    // fonctionner dans une postcondition d'une methode statique,
    // d'ou la non-utilisation de approxEgal.

    @Pre( "x >= 0.0" )
    @Post( "Math.abs( @Result * @Result - x ) <= eps" )
    public static double sqrt( double x ) {
      return Math.sqrt( x );
    }


    //
    // Un petit programme de test.
    //
     public static void main( String[] args ) {
        double x1 = 2.0;
        System.out.println( "sqrt( " + x1 + " ) = " + sqrt( x1 ) );

        double x2 = -1.0;
        System.out.println( "sqrt( " + x2 + " ) = " + sqrt( x2 ) );
    }
}
```

```
sqrt( 2.0 ) = 1.4142135623730951
Exception in thread "main" jass.modern.core.PreConditionError: [x >= 0.0]
        at Exemples.Sqrt.sqrt(Sqrt.java:16)
        at Exemples.Sqrt.main(Sqrt.java:24)
```

Question : Si on supprime la précondition et qu'on garde tout le reste du programme identique, l'effet à l'exécution est alors le suivant :

```
sqrt( 2.0 ) = 1.4142135623730951
Exception in thread "main" jass.modern.core.PostConditionError:
    [Math.abs( _Return * _Return - x ) <= eps]
        at Exemples.Sqrt4.sqrt(Sqrt4.java:15)
        at Exemples.Sqrt4.main(Sqrt4.java:23)
```

Pourquoi? (Indice : Qu'est-ce qui est imprimé si on supprime aussi la postcondition?)

- Une spécification semblable à celle de l'exemple JML — où on indique explicitement dans le contrat qu'une exception spécifique doit être signalée — aurait l'allure suivante :

```
@Also({
 @SpecCase( pre = "x >= 0.0",
            post = "Math.abs( @Result * @Result - x ) <= eps" ),
 @SpecCase( pre = "x < 0.0",
            signals = RacineImaginaire.class )
})
```

Dans ce cas, la mise en oeuvre de `sqrt` serait alors différente, style :

```
if ( x <= 0.0 ) {
  throw new RacineImaginaire();
}
return Math.sqrt( x );
```

6.2 Une classe d'objets pour des piles bornées

Les exemples 8 et 9 présentent deux variantes d'une spécification pour une classes d'objets pour des *piles bornées* :

- La visibilité de l'attribut `tailleMax` est `public`. Ceci est nécessaire avec l'outil Modern JASS car cet attribut est utilisé dans les contrats.
- Certaines fonctions sont annotées comme étant `@Pure`. La signification est celle décrite dans le chapitre sur la spécification de contrats pour des fonctions (en OCL) : il s'agit de méthodes qui représentent des *fonctions pures*, donc qui n'ont aucun effet de bord.
- On remarque que certaines méthodes n'ont pas de contrat, par exemple, `nbElements` : il s'agit donc d'une requête primitive, au sens vu précédemment (chapitre OCL sur la spécification de types abstraits).
- L'exemple 8 est une version des piles sans exceptions explicites, donc semblable à la version OCL vue précédemment, où c'est uniquement le comportement «normal» qui est décrit. Donc, si une précondition n'est pas satisfaite, alors *le comportement est indéfini.*

Exemple 8 Spécification Modern JASS d'une classe d'objets pour des piles bornées sans exceptions explicites.

```
public class Pile0<T> {
  @Invariant( "nbElems <= tailleMax" )

  protected T[] elems;
  public int tailleMax;
  protected int nbElems;


  @Post( "estVide()" )
  @SuppressWarnings("unchecked")
  public Pile0( int tailleMax ) {
    this.tailleMax = tailleMax;
    elems = (T[]) new Object[tailleMax];
    nbElems = 0;
  }


  @Pre ( "nbElements() < tailleMax" )
  @Post( "nbElements() == @Old(nbElements())+1 && sommet() == e" )
  public void empiler( T e ) {
      elems[nbElems] = e;
      nbElems++;
  }

  @Pre ( "!estVide()")
  @Post( "nbElements() == @Old(nbElements())-1" )
  public void depiler() {
      nbElems--;
  }

  @Pre( "!estVide()")
  public T sommet() {
      return elems[nbElems-1];
  }

  @Post( "@Result == (nbElements() == 0)" )
  @Pure public boolean estVide() {
    return nbElems == 0;
  }

  @Pure public int nbElements() {
    return nbElems;
  }
}
```

- L'exemple 9, quant à lui, est une version des piles *avec exceptions spécifiques explicites*. Le comportement attendu de chacune des opérations est décrit de façon *complète*, y compris dans les situations «anormales». En d'autres mots, le comportement est complètement spécifié pour tous les états et toutes les valeurs possibles : soit le comportement est «normal», soit une exception spécifique est signalée. Comme on le voit, c'est la construction `@Also` qui permet de spécifier les différentes alternatives possibles, chaque cas étant décrit par un `@SpecCase` avec ses propres pré/postconditions.

6.3 Une classe d'objets pour des dictionnaires (non bornés)

L'exemple 10 présente la spécification pour un type mutable (classe d'objets) `Dictionnaire` semblable à celui vu en OCL :

- La mise en oeuvre d'un dictionnaire est réalisée à l'aide des collections, immuables, de la bibliothèque `OclCollections`.
- Un dictionnaire est représenté par un ensemble de paires clé/définition. Cette notion de paire est réalisée par la classe auxiliaire `PaireCleDefn` — Classe Java 1.
- La méthode auxiliaire `defnsEgales` est requise parce qu'un appel de méthode apparaissant dans une pré/postcondition doit nécessairement être un appel à une fonction pure. Or, la méthode `equals` étant définie dans une autre classe (méthode du type générique `D`), le compilateur ne sait pas qu'elle est pure. En introduisant une méthode auxiliaire annotée avec `@Pure`, on règle ainsi le problème.
- La postcondition est décrite à l'aide de différentes clauses (`@Also`), chacune décrivant une des propriétés du résultat ou de l'effet produit par la méthode.
- Pour indiquer la valeur d'un attribut ou d'une méthode avant l'appel, on utilise `@Old`. On remarque qu'il s'agit ici d'une sorte de fonction, qui reçoit un élément en argument — contrairement à OCL où on utilise l'opérateur `@pre` en suffixe.

7 Exercice : Exemple Eiffel

Le langage Eiffel est le tout premier langage de programmation à avoir introduit la possibilité de spécifier des contrats et de les vérifier dynamiquement (à l'exécution) [Mey88]. Eiffel fut développé par Bertrand Meyer, qui a popularisé l'approche DBC (*Design By Contract*) [Mey85, Mey92].

Plus précisément, de nombreux autres langages supportent maintenant la vérification dynamique d'assertions, et dans certains cas la spécification et la vérification dynamique de contrats. Par contre, Eiffel reste le seul langage où ces aspects sont intégrés *directement dans le langage*, et non comme des ajouts ultérieurs exprimés à l'aide de commentaires spéciaux ou d'annotations traités par des préprocesseurs.

L'exemple 11[10] présente la spécification d'une classe pour un type `SIMPLE_X` en Eiffel.

Quelques éléments du langage Eiffel :

- Les crochets «[...]» après le nom de la classe indique qu'il s'agit d'une classe générique, d'argument générique `T` (nécessairement un type).
- La clause **creation** indique laquelle des méthodes (appelées **feature**s en Eiffel) est utilisée pour créer un nouvel objet. Dans cet exemple, il s'agit de la **feature** *initialize*, qui décrit donc l'état initial d'un objet nouvellement créé — cette spécification d'un type décrit donc une *classe d'objets* (un type mutable).

[10] Légèrement adapté de celui présenté dans «Design by Contract, by Example» [MM02, Chap. 2].

Exemple 9 Spécification Modern JASS d'une classe d'objets pour des piles bornées avec exceptions explicites.

```
public class Pile1<T> {
  @Invariant( "nbElems <= tailleMax" )

  protected T[] elems;
  public int tailleMax;
  protected int nbElems;

  @Post( "estVide()" )
  @SuppressWarnings("unchecked")
  public Pile1( int tailleMax ) {
    this.tailleMax = tailleMax;
    elems = (T[]) new Object[tailleMax];
    nbElems = 0;
  }

  @Also({ @SpecCase( pre = "nbElements() < tailleMax",
                     post = "nbElements() == @Old(nbElements())+1 && sommet() == e" ),
          @SpecCase( pre = "nbElements() == tailleMax",
                     signals = PilePleine.class ) })
  public void empiler( T e ) throws PilePleine {
    if ( nbElems < tailleMax ) {
      elems[nbElems] = e; nbElems++;
    } else {
      throw new PilePleine();
    }
  }

  @Also({ @SpecCase( pre = "!estVide()",
                     post = "nbElements() == @Old(nbElements())-1" ),
          @SpecCase( pre = "estVide()",
                     signals = PileVide.class ) })
  public void depiler() throws PileVide {
    if ( nbElems > 0 ) {
      nbElems--;
    } else {
      throw new PileVide();
    }
  }

  @Pre( "!estVide()" )
  @SpecCase( pre = "estVide()", signals = PileVide.class )
  public T sommet() throws PileVide {
    if ( nbElems > 0 ) {
      return elems[nbElems-1];
    } else {
      throw new PileVide();
    }
  }

  @Post( "@Result == (nbElements() == 0)" )
  @Pure public boolean estVide() {
    return nbElems == 0;
  }

  @Pure public int nbElements() {
    return nbElems;
  }
```

Exemple 10 Spécification Modern JASS pour des dictionnaires non bornés.

```
public class Dictionnaire<C,D> {
    public Set<Paire<C,D>> elems;

    @Post( "elems.isEmpty()" )
    public Dictionnaire() {
      elems = Collections.emptySet();
    }

    @Pure public boolean estDefinie( C cle ) {
      return pairePour( cle ) != null;
    }

    @Pre( "estDefinie(cle)" )
    @Pure public D definitionPour( C cle ) {
      return pairePour( cle ).defn();
    }

    @Also({  @SpecCase( post = "estDefinie(cle)" ),
             @SpecCase( post = "defnsEgales( definitionPour(cle), defn )"),
             @SpecCase( post = "@Old(nbElements()) <= nbElements() &&" +
                               "nbElements() <= @Old(nbElements())+1" )
    })
    public void definir( C cle, D defn ) {
      Paire<C,D> p = pairePour( cle );
      if ( p != null ) { elems = elems.excluding( p ); }
      elems = elems.including( new PaireCleDefn<C,D>( cle, defn ) );
    }

    @Pre( "estDefinie(cle)" )
    @Post( "!estDefinie(cle) && nbElements() == @Old(nbElements()) - 1" )
    public void supprimer( C cle ) {
      Paire<C,D> p = pairePour( cle );
      if ( p != null ) { elems = elems.excluding( p ); }
    }

    @Pure public int nbElements() {
      return elems.size();
    }

    private Paire<C,D> pairePour( final C cle ) {
      for( Paire<C,D> p : elems ) {
         if ( p.cle().equals(cle) ) { return p; }
      }
      return null;
    }

    @Pure public boolean defnsEgales( D d1, D d2 ) {
      return d1.equals(d2);
    }
}
```

Classe Java 1 Classe auxiliaire pour représenter des paires clé/définition.

```
class PaireCleDefn<C,D> implements Paire<C,D> {
    private C cle;
    private D defn;

    PaireCleDefn( C cle, D defn ) {
      this.cle = cle;
      this.defn = defn;
    }

    public C cle() {
      return cle;
    }

    public D defn() {
      return defn;
    }

    @SuppressWarnings( "unchecked" )
    @Pure public boolean equals( Object o ) {
      Paire<C,D> p = (Paire<C,D>) o;
      return cle.equals( p.cle() );
    }
}
```

- Une **feature** représente une méthode publique, exportée par le module (par la classe).
- Une **feature** telle que «*item_at* `( i: INTEGER ): T`» dénote une opération qui prend un argument entier et qui retourne un résultat de type `T`.
- Comme en iContract, le mot-clé **require** indique une précondition, alors que le mot-clé **ensure** indique une postcondition.
- L'identificateur suivi immédiatement d'un «:» qui précéde la condition d'une pré/post-condition sert essentiellement à nommer et à documenter la condition — comme en OCL. Entre autres, c'est cet identificateur qui sera affiché si, lors l'exécution, la condition associée n'est pas satisfaite.
- Le style de spécification est semblable à celui de `iContract` : on ne spécifie pas de modèle explicite ; on spécifie plutôt l'effet d'une **feature** sur certains observateurs primitifs.

Exercice : Quel type abstrait cette spécification décrit-elle?

Exemple 11 Spécification d'une classe générique `SIMPLE_X[T]` en Eiffel (adapté de [MM02]).

```
class interface
   SIMPLE_X[T]

creation
   initialize

feature
   count: INTEGER

   item_at( i: INTEGER ): T
     require
       i_big_enough: i >= 1;
       i_small_enough: i <= count

   is_empty: BOOLEAN
     ensure
       consistent_with_count: Result = (count = 0)

   item: T
     require
       x_not_empty: count > 0
     ensure
       consistent_with_item_at: Result = item_at(count)

   initialize
     ensure
       x_is_empty: count = 0

   put( x: T )
     ensure
       count_increased: count = old count + 1
       x_on_p: item_at(count) = x

   remove
      require
         x_not_empty: count > 0
      ensure
         count_decreased: count = old count - 1

invariant
  count_is_never_negative: count >= 0

end
```

A Autres utilisations des assertions

Dans un chapitre précédent, nous avons vu que les assertions pouvaient être utilisées lors des tests en Java, avec JUnit, plus précisément pour la spécification des résultats attendus.

Des sections qui précèdent, nous pouvons voir que la vérification dynamique des contrats représente aussi une forme d'utilisation des assertions :

- Une précondition peut être réalisée par une assertion qui apparaît au tout début d'une méthode ;
- Une postcondition peut être réalisée par une assertion qui apparaît juste avant de terminer l'exécution d'une méthode, donc juste avant de retourner le résultat lorsque cette méthode en produit un.
- Un invariant est une condition qui doit être vraie en tout temps. Puisqu'il est impossible de constamment vérifier la condition associée, on peut choisir d'évaluer une assertion qui représente l'invariant à la fin de chaque méthode, pour assurer que l'exécution de la méthode a bien préservé cet invariant.

D'autres utilisations des assertions sont aussi suggérées, telles que décrites dans la documentation Java et de laquelle sont extraits les exemples qui suivent[11] :

- Pour spécifier des invariants internes, liés spécifiquement à la mise en oeuvre.

> *Before assertions were available, many programmers used comments to indicate their assumptions concerning a program's behavior. For example, you might have written something like this to explain your assumption about an else clause in a multiway if-statement:*

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // We know (i % 3 == 2)
    ...
}
```

> *You should now use an assertion whenever you would have written a comment that asserts an invariant. For example, you should rewrite the previous if-statement like this:*

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

[11] *Programming with assertions*, `http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`

- Pour assurer explicitement qu'une branche `default` d'une instruction `switch` ne peut pas s'exécuter :

```
switch(suit) {
  case Suit.CLUBS:
    ...
    break;

  case Suit.DIAMONDS:
    ...
    break;

  case Suit.HEARTS:
    ...
    break;

  case Suit.SPADES:
    ...
  default:
    assert false : suit;
}
```

- Pour assurer qu'un bout de code qui n'est pas supposé s'exécuter ne s'exécute pas :

```
 void foo() {
    for (...) {
        if (...)
            return;
    }
    assert false; // Execution should never reach this point!!!
}
```

Règle importante concernant les assertions

Une règle importante concernant les assertions est que les expressions utilisées dans les assertions *ne devraient jamais avoir d'effets de bord* — ne devraient jamais modifier d'autres variables ou effectuer des entrées/sorties. La raison en est bien simple : par défaut, en Java, l'évaluation dynamique des assertions est désactivée, et donc si l'option d'évaluation n'est pas explicitement activée, un programme contenant des effets de bord dans les assertions pourrait ne pas avoir le même comportement que si les assertions était activées.

Une exception à cette règle est celle décrite dans la documentation Java, bout de code qui permet de terminer l'exécution d'un programme si les assertions ne sont pas activées :

```
static {
    boolean assertsEnabled = false;
    assert assertsEnabled = true; // Intentional side effect!!!
    if (!assertsEnabled)
        throw new RuntimeException("Asserts must be enabled!!!");
}
```

Références

[BCC+05] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[Ens01] O. Enseling. iContract: Design by contract in Java. *Java World*, February 2001. `http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html`.

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

[LCC+03] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):262–284, March 2003.

[LG01] B. Liskov and J. Guttag. *Program Development in Java.* Addison-Wesley, 2001.

[Mey85] B. Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, Jan. 1985.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction (Second edition).* Prentice-Hall, 1997.

[MM02] R. Mitchell and J. McKim. *Design by Contract, by Example.* Addison-Wesley, 2002.

[Rie07] J. Rieken. Design by contract for java — revised. Master's thesis, Universitat Oldenburg, Department fur Informatik, April 2007.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language—Second Edition: Getting Your Models Ready for MDA.* Addison-Wesley, 2003.