Université de Montpellier / Faculté Des Sciences

Modélisation et Programmation par Objets (2e partie)

2019-2020 - HLIN 505

Cours 3
Programmation par contrats
Assertions
Exceptions

Enseignants

Marianne Huchard, Stéphane Bessy, Marie-Laure Mugnier

Clémentine Nebut, Abdelhak-Djamel Seriai

Ce document est un recueil de notes de cours. Il peut développer des aspects vus en cours plus succinctement ou inversement vous aurez en cours plus de détails ou d'autres exemples sur certains points. Si vous y relevez des erreurs, n'hésitez pas à nous les signaler afin de l'améliorer.

1 La conception et la programmation par contrats

Par conception et programmation par contrats, on entend la mise en place de règles sur les classes et les méthodes et leur vérification. Ces règles établissent des contrats entre les objets (sur leurs responsabilités respectives) qui collaborent pendant l'exécution d'un programme à objets. Cette notion a été mise en avant dans les approches à objets par Bertrand Meyer et mise en œuvre de manière native dans le langage Eiffel. Elle est développée de différentes manières suivant les langages : elle peut être intégrée directement dans le langage lui-même ou être intégrée dans des outils de développement annexes. Il s'agit d'expliciter des spécifications et de réduire les erreurs pendant le développement ou pendant l'exécution.

Les contrats portent principalement sur :

— les invariants de classes (assertions sur les état des objets) ou d'algorithmes (propriétés vérifiées en certains points internes des méthodes).

— les pré-conditions (propriétés vérifiées à l'entrée d'une méthode, avant l'appel). Une pré-condition est une responsabilité du client de la méthode.

— les post-conditions (propriétés vérifiées à la sortie d'une méthode, après l'appel). Une post-condition est une responsabilité de la méthode envers son client.

Java possède deux principales techniques de mise en place de contrats pour les classes : les assertions et les exceptions. Nous les étudions dans ce cours au travers de l'étude du type abstrait Pile.

La conception par contrat ne se résume bien évidemment pas aux assertions et aux exceptions, elle peut être traitée par des spécifications, des preuves formelles ou de la vérification de modèle (*model checking*). La conception et la réalisation des tests y font également référence.

Le plan du cours est le suivant :

— Le type abstrait *Pile* et une structure de données qui le met en œuvre,
— Les assertions qui servent à la mise au point du type abstrait.
— Les exceptions qui servent à sécuriser son usage.

2 Le type abstrait *Pile*

Le type abstrait *Pile* se définit ainsi.

— le type défini est *Pile*
— il est paramétré par un autre type T

Opérations Les opérations autorisées sur une pile sont :

— initialiser()
— empiler(T element)
— depiler() : T
— sommet() : T
— estVide() : boolean

Préconditions On peut a priori toujours initialiser, empiler et savoir si la pile est vide. Par contre on ne peut dépiler ou consulter le sommet d'une pile que si elle est non vide. Donc si p est une pile :

— précondition (p.depiler()) : $\neg\ p.estVide()$
— précondition (p.sommet()) : $\neg\ p.estVide()$

Axiomes si p est une pile, et t un élément de type T, les axiomes indiquent le fonctionnement des opérations.

1. Juste après l'instruction $p.empiler(t)$, on a $p.sommet() = t$
2. Juste après $p.initialiser()$, on a $p.estVide() = true$
3. Juste après $p.empiler(t)$, on a $p.estVide() = faux$
4. Juste après $p.empiler(t)$, on a $p.depiler() = t$

Nous proposons de représenter le type abstrait *Pile* par une interface. Comprenez l'interface comme une classe où toutes les méthodes sont abstraites et notez le paramètre formel de généricité `<T>` qui indique que l'on peut réaliser des piles de tous les types d'éléments que l'on veut (mais une pile donnée contient des éléments du même type). Ces notions feront l'objet d'un prochain cours détaillé.

```
public interface IPile<T>
{
   void initialiser();
   void empiler(T t);
   T depiler();
```

```
   T sommet();
   boolean estVide();
}
```

Puis cette interface est implémentée en Java par une classe. Cette classe contient une fonction *main* qui illustre le fonctionnement de la pile.

```
public class Pile<T> implements IPile<T>{
   private ArrayList<T> elements;

   public Pile(){initialiser();}

   public T depiler() {
     T sommet = elements.get(elements.size()-1);
     elements.remove(sommet);
     return sommet;
   }

   public void empiler(T t) {elements.add(t);}

   public boolean estVide() {return elements.isEmpty();}

   public void initialiser() {elements = new ArrayList<T>();}

   public T sommet() {return elements.get(elements.size()-1);}

   public String toString(){return "Pile = "+ elements;}

   public static void main(String[] a)
   {
      Pile<String> p = new Pile<String>();
      System.out.println(p);
      p.empiler("a"); p.empiler("b"); p.empiler("c");
      System.out.println(p);
      p.depiler();
      System.out.println(p);
      p.depiler(); p.depiler();
      System.out.println(p);
    }
}
```

3 Assertions en Java

Les assertions sont utilisées principalement en phase de mise au point des programmes et permettent de vérifier des propriétés sur une classe et plus particulièrement :

— les invariants de classe (par exemple, *une voiture a 4 roues, l'âge d'une personne est positif*).

— les invariants d'algorithmes, les axiomes des types abstraits de données (par exemple, *à la fin de l'itération i dans la recherche du maximum dans un tableau, la variable max contient la plus grande valeur rencontrée entre 0 et i*).
— les invariants de flux (par exemple, *ce point du programme ne peut être atteint*).
— les post-conditions (par exemple, *à la fin d'un tri de tableau, le tableau est trié*).

Le programme est interrompu en cas de non respect de ces propriétés. Les assertions peuvent être activées ou inhibées et sont principalement une aide au débogage. Pour les activer (resp. les désactiver), il faut exécuter l'interprète *java* avec l'option *−ea* (resp. *−da*). Sous eclipse, il faut changer les arguments de la configuration d'exécution, comme le montre la figure 1.

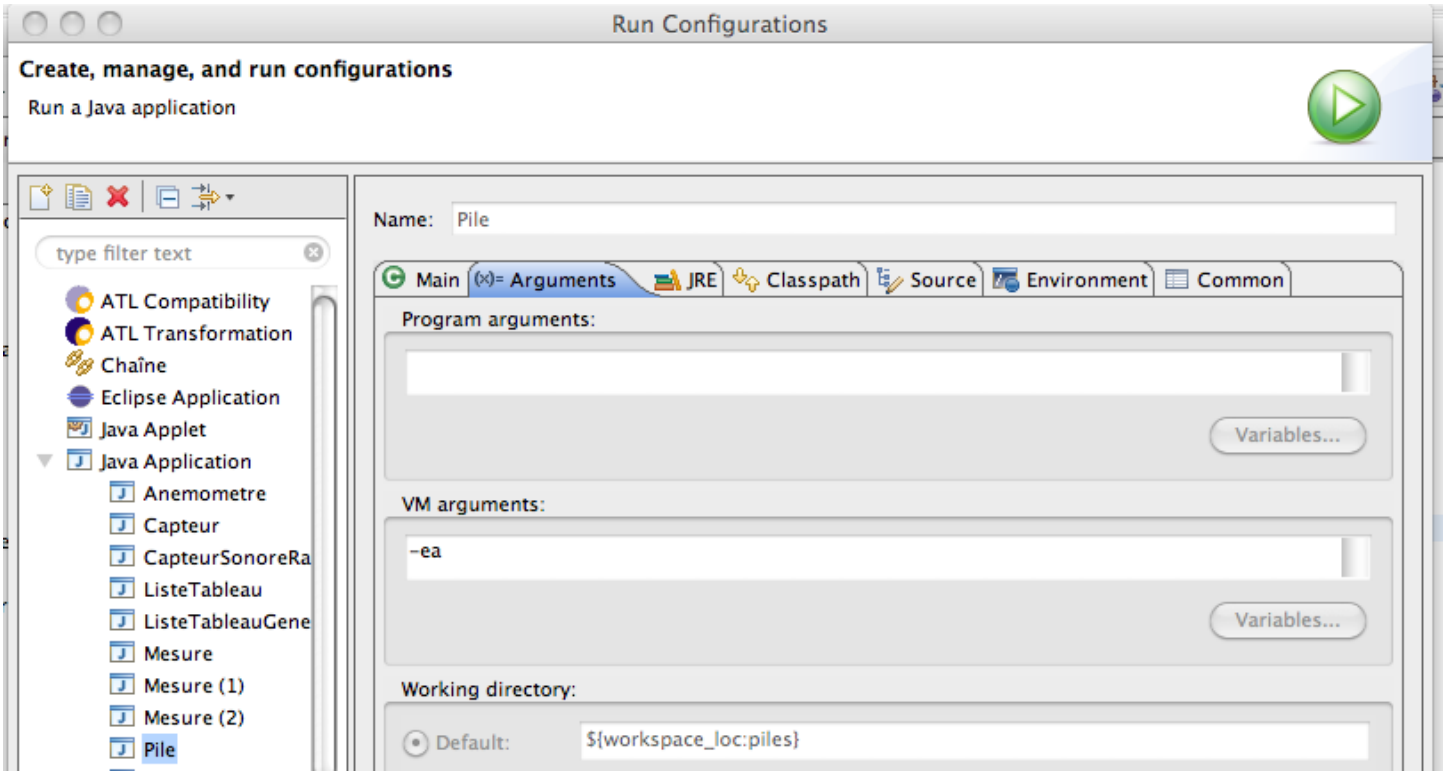

FIGURE 1 – Configuration sous eclipse pour activer les assertions

Deux syntaxes sont possibles :

```
assert condition ;
assert condition : objet ;
```

Par exemple, pour vérifier le premier axiome qui indique qu'après avoir empilé un élément t dans une pile, cet élément apparaît au sommet, on peut écrire l'assertion en fin de la méthode.

```
public void empiler(T t) {
   elements.add(t);
   assert this.sommet()==t;
}
```

Supposons que l'on écrive une version erronée de la méthode *empiler*, telle que l'ajout de *null* à la fin du conteneur des éléments de la pile :

```
public void empiler(T t) {
   elements.add(t); elements.add(null);
   assert this.sommet()==t ;
}
```

L'exécution du *main* précédent s'arrêtera et vous obtiendrez le message suivant :

```
Exception in thread "main" java.lang.AssertionError
at pile.Pile.empiler(Pile.java:20)
at pile.Pile.main(Pile.java:48)
```

Si l'on veut être plus précis, on peut mettre un objet dans l'assertion, l'erreur affichée mentionnera le résultat de l'appel de *toString*() sur cet objet.

```
public void empiler(T t) {
   elements.add(t);
   assert this.sommet()==t : "dernier empile ="+this.sommet();
}
```

Cette fois l'erreur affichée sera la suivante.

```
Exception in thread "main" java.lang.AssertionError: dernier empile =null
at pile.Pile.empiler(Pile.java:20)
at pile.Pile.main(Pile.java:48)
```

4 Exceptions en Java

On utilisera le mécanisme de gestion des exceptions pour signaler mais aussi pour rattraper des erreurs pouvant se produire pendant l'exécution du programme. A l'inverse du mécanisme d'assertions qui sert principalement à la mise au point, le mécanisme de gestion des exceptions sert à éviter des arrêts brutaux du programme, et à revenir dans un fonctionnement normal, ou à stopper le programme proprement, après avoir éventuellement sauvegardé des données, fermé des fichiers, etc.

En Java, les exceptions sont des objets qui représentent une erreur à l'exécution correspondant :

- quelquefois aux invariants de classe.
- aux pré-conditions : lorsqu'une méthode est appelée hors de son domaine de définition (par exemple, *dépiler une pile vide, ou donner une mauvaise valeur de paramètre*).
- certains problèmes ou post-conditions que l'on peut traiter à l'exécution
  - une faute de saisie (par exemple, *saisir "aa" quand on attend un entier*),
  - un problème matériel (par exemple, *plus de place lors de l'écriture d'un fichier*),
  - une faute de programmation (par exemple, *sortir des bornes d'un tableau, une division par zéro*).

Il y a un certain recouvrement entre ce que l'on traitera par des assertions et ce que l'on traitera par des exceptions, cela reste donc un choix de conception et de programmation.

En Java, l'API contient de nombreuses représentations d'erreurs, parmi les plus courantes quand on débute en programmation :

- `NullPointerException` (on a appelé `o.f()` mais `o` vaut `null`).
- `ArrayIndexOutOfBoundException` (on est sorti des bornes du tableau).

La classe `Throwable` est la classe mère de toutes les exceptions (voir figure 2). Elle contient les éléments suivants :

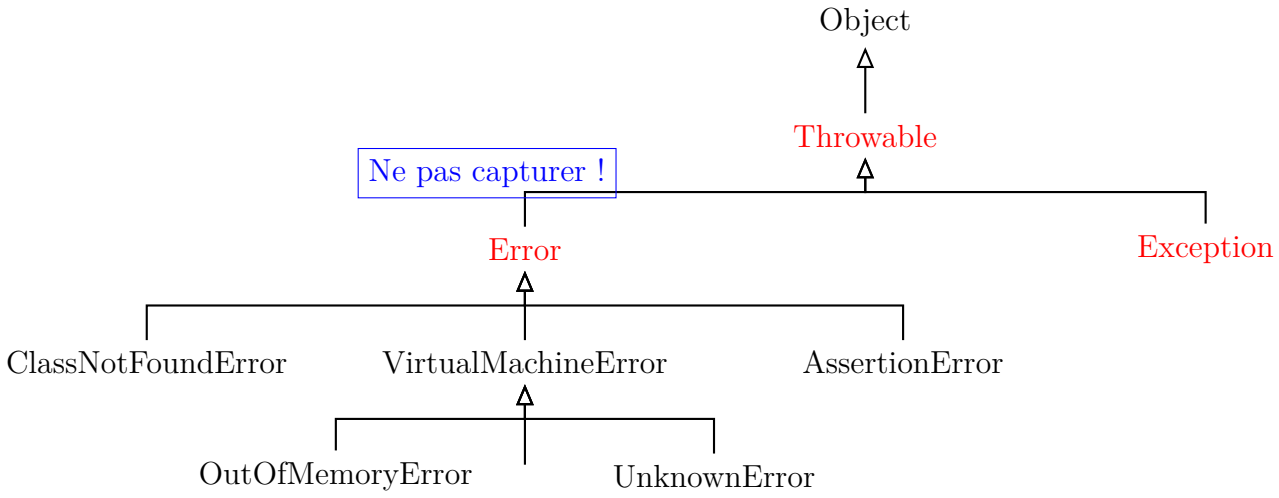

FIGURE 2 – Premiers étages de la hiérarchie des exceptions en Java

- attributs
  - un message d'erreur rapportant l'erreur (une String)
  - l'état de la pile des appels de méthodes
- méthodes
  - public Throwable()
  - public Throwable(String message)
  - public String getMessage()
  - public void printStackTrace() (imprime la pile des appels de méthodes et permet de comprendre dans quel contexte s'est produite l'erreur)

On ne capturera généralement pas les `Error` car elles correspondent aux problèmes de la machine virtuelle, incluant les manques de ressources ou les assertions non respectées (Figure 2). Une autre partie importante à connaître est la classe `Exception` qui correspond aux exceptions définies par le programmeur d'une API ou d'un programme final (Figure 3). Parmi les sous-classes de la classe `Exception`, la classe `RunTimeException` joue un rôle particulier. Ses instances, comme celles de la classe `Error`, n'auront pas besoin d'être *déclarées explicitement dans les signatures des méthodes*. Cette notion de déclaration est détaillée dans la suite.

Définition d'une classe d'erreur sur une pile Dans le cas de la classe `Pile` telle que nous l'avons esquissée, une erreur peut ainsi se produire lors de l'appel des opérations *dépiler* ou *sommet* sur une pile vide. Cette erreur va se représenter à l'aide d'une instance de la classe `PileVideException` que nous définissons. Cette classe spécialise la classe de l'API Java `Exception`. C'est nécessaire pour bénéficier du mécanisme d'exceptions. On note que cette classe n'est pas très compliquée, elle pourrait avoir des attributs décrivant la localisation de l'erreur ou l'état des objets au moment où l'erreur s'est produite, et des opérations de traitement de l'erreur.

```
public class PileVideException extends Exception {
    public PileVideException() {  }
    public PileVideException(String message) { super(message); }
}
```

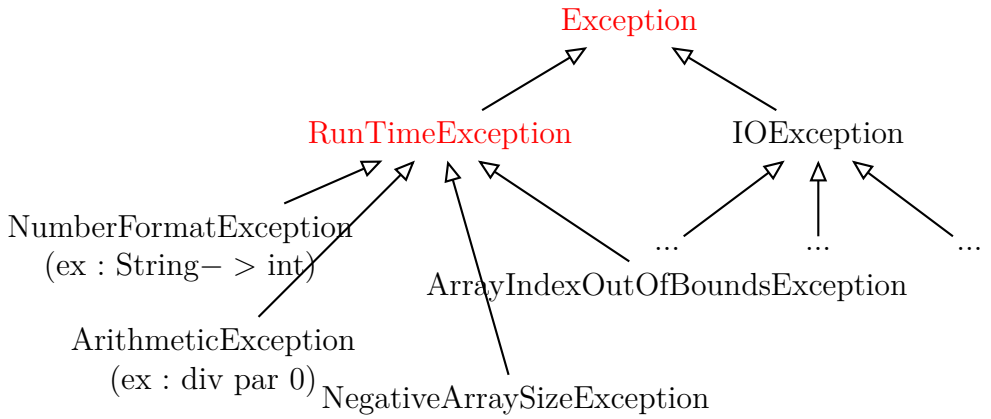

FIGURE 3 – Suite de la hiérarchie des exceptions en Java (sous-classes de `Exception`)

Déclaration et signalement d'une erreur Cette classe d'exception nous permet de réécrire une version plus sécurisée de la méthode `dépiler`. Il faut noter dans la signature (première ligne) la **déclaration** du fait que la méthode risque de s'interrompre à cause de l'erreur *PileVideException* et le **signalement** effectué dans le code lorsque l'on détecte que l'on essaie de dépiler une pile vide.

```
public T depiler() throws PileVideException{   // déclaration
    if (this.estVide())
        throw new PileVideException("en dépilant");  // signalement
    T sommet = elements.get(elements.size()-1);
    elements.remove(sommet);
    return sommet;
}
```

Notez que cela oblige à prévoir le signalement dans l'interface également qui sera modifiée en conséquence.

```
public interface IPile<T>
{
....
   T depiler() throws PileVideException;
...
}
```

Règles de substituabilité de signature Avec l'ajout des exceptions dans les signatures des méthodes, se repose la question des règles de redéfinition des méthodes pour assurer la liaison dynamique. Ces règles respectent les mêmes principes que les règles portant sur les listes de types de paramètres ou de type de retour : le système de vérification des types ne veut rien d'inattendu.

Dans une redéfinition d'une méthode :

— Ajouter une déclaration n'est pas possible,

— Retirer une exception est possible,

— Spécialiser une exception est possible.

Par exemple, supposons que l'on dispose de la classe d'exception `PileException` (exceptions portant sur les méthodes d'une classe représentant les piles) et de sa sous-classe `RecuperationPileException`. Une méthode sur les piles dont la signature serait dans une classe `Pile` :

```
recuperer() throws PileException, IOException
```

peut être redéfinie dans une sous-classe `PileBornee` de `Pile` en :

```
recuperer() throws RecuperationPileException.
```

L'exception `IOException` a été retirée, et l'exception `PileException` a été spécialisée par `RecuperationPileException`.

Récupération d'une erreur Le programme suivant va tout de même s'interrompre brutalement avec une erreur et n'atteindra pas la fin.

```
public static void main(String[] a)
{
    Pile<String> p = new Pile<String>();
    System.out.println(p);
    p.empiler("a"); p.empiler("b"); p.empiler("c");
    System.out.println(p);
    p.depiler();
    System.out.println(p);
    p.depiler(); p.depiler();  p.depiler();
    System.out.println(p);
}
```

Pour éviter cet arrêt brutal, on utilise un bloc de contrôle particulier, le bloc `try - catch` de syntaxe :

```
try{BLOC-0}
catch (Class-1 e1){BLOC-1}
...
catch(Class-n en){BLOC-n}
finally{BLOC-n+1}
```

Lorsqu'une instance d'exception est générée dans un bloc *try*, l'exécution de `BLOC-0` s'interrompt. Les clauses *catch* sont examinées dans l'ordre, jusqu'à en trouver une (notons-la `Class-i`) qui déclare une classe à laquelle appartient l'instance d'exception. S'il en existe une, le bloc du *catch* associé (ici `BLOC-i`) est exécuté, et l'exécution reprend juste après les clauses *catch*. Sinon, l'exception n'est pas capturée, elle est donc transmise à l'appelant (une autre méthode et/ou un autre bloc try/catch), et le reste de la méthode n'est pas exécuté. On passe dans tous les cas par le bloc *finally*, dans lequel on pourra désallouer des ressources (fermer des fichiers par exemple) ou avoir toute action qui rendrait aussi cohérent que possible l'état du programme.

Le code ci-dessous vous montre une utilisation simple d'un tel bloc.

```
public static void main(String[] a)
{
    try{
        Pile<String> p = new Pile<String>();
```

```
        System.out.println(p);
        p.empiler("a"); p.empiler("b"); p.empiler("c");
        System.out.println(p);
        p.depiler();
        System.out.println(p);
        p.depiler(); p.depiler();  p.depiler();
        System.out.println(p);
    }
    catch (PileVideException p){System.out.println(p.getMessage());}
    System.out.println("Fin de la méthode main");
}
```

Nous examinons maintenant un cas plus complexe de l'utilisation de ce bloc de récupération des erreurs. Nous considérons 4 classes d'exception avec une relation d'héritage entre certaines de ces classes.

```
public class E1 extends Exception {}
public class E2 extends Exception {}
public class E3 extends E2 {}
public class E4 extends Exception {}
```

Nous allons nous préoccuper de ce qu'affiche le programme ci-après suivant l'exception signalée dans `meth1`.

```
public class TestException {

    public void meth1() throws E1, E2, E3, E4
    {
        .......... throw .....
    }

    public void meth2() throws E4
    {
        try{
            System.out.print("(1)");
            meth1();
        }
        catch(E1 e){System.out.print("(2)");}
        catch(E2 e){System.out.print("(3)");}
        finally {System.out.print("(4)");}
        System.out.println("(5)");
    }

    public static void main(String[] arg) throws E4
    {
        TestException t = new TestException();
        t.meth2();
    }
```

```
}
```

Le résultat est le suivant (`exerciceCours2011` est le nom du *package*) :
- s'il n'y a rien de signalé dans meth1 : (1)(4)(5)
- avec throw new E1() ; dans meth1 : (1)(2)(4)(5)
- avec throw new E2() ; dans meth1 : (1)(3)(4)(5)
- avec throw new E3() ; dans meth1 : (1)(3)(4)(5)
- avec throw new E4() ; dans meth1 : (1)(4)Exception in thread "main" exerciceCours2011.E4
  at exerciceCours2011.TestException.meth1(TestException.java :7)
  at exerciceCours2011.TestException.meth2(TestException.java :14)
  at exerciceCours2011.TestException.main(TestException.java :25)

Try avec ressource attachée Depuis Java 1.7, on peut définir des ressources directement au début (et attachées) à l'instruction `try`. La ressource est alors automatiquement fermée à la fin de l'exécution du bloc `try`. Par exemple, dans le code suivant, `bufferedReader` sera fermé proprement à la fin normale ou anormale des traitements par un appel automatique à `close`.

```
try (BufferedReader bufferedReader =
     new BufferedReader(new FileReader("myFile.txt")))
{
        String line=null;
        while ((line = bufferedReader.readLine()) != null)
        {
                        System.out.println(line);
        }
}
catch (IOException ioe) {
        ioe.printStackTrace();
}
```

À un `try` on peut attacher tout objet implémentant `java.lang.AutoCloseable`.
`java.lang.AutoCloseable` définit une seule méthode, `close()`, qui lève une exception de type Exception.
Enfin on peut spécifier plusieurs ressources attachées :
`try (Ressource a=...; Ressource b=...; Ressource c=...)`
À la fin, c, puis b, puis a seront fermées automatiquement.

Capture de plusieurs exceptions dans un même `catch` C'est une autre nouveauté de la version 1.7 de Java, avec la syntaxe suivante appliquée à un exemple :

```
try {  ... }
catch(IOException | SQLException  ex) {  ex.printStackTrace();  }
```

Schémas de retransmission des erreurs Dans le cadre de la gestion des erreurs, on observera des survenues d'exceptions de bas niveau que l'on ne traitera pas, mais que l'on retransmettra (parce que l'on ne sait pas les traiter). Elles peuvent être :
- retransmises telles quelles. Dans ce cas on ne fait pas de nouveau signalement ; si ce ne sont pas des `Error` ni des `RunTimeException` on les déclare simplement dans la signature de l'opération pour éviter une erreur de compilation.

— capturées pour transmettre une erreur d'un niveau d'abstraction plus élevé.

Par exemple, supposons que l'on ait une méthode de récupération de l'état de la pile dans un fichier `recuperer(String filename)` dont on donne le nom en paramètre. Si le fichier n'existe pas, une exception `FileNotFoundException` se produit au sein de la méthode. Cette exception peut être :

— ou bien transmise sans modification par la méthode de récupération,

— ou bien capturée par cette méthode de récupération, qui à la place, signalera une exception `RecuperationPileImpossibleException` (avec un attribut ou un message indiquant l'origine du problème). Cette exception de type `RecuperationPileImpossibleException` sera une instance d'une classe écrite par le programmeur, dont le niveau d'abstraction (conceptuelle) est le même que celui de la classe `Pile`. Elle fera partie des contrats de la pile.

5 Synthèse

Nous rappelons les bonnes et les mauvaises pratiques pour l'usage des exceptions et des assertions.

Il est surtout conseillé d'utiliser les assertions lors de la mise au point pour exprimer les propriétés suivantes :

— invariant d'algorithme, de flux,

— post-conditions,

— invariants de classe.

Il est surtout conseillé d'utiliser les exceptions pour la récupération d'erreurs à l'exécution :

— vérifier les pré-conditions et en particulier vérifier les paramètres d'une méthode car ces paramètres viennent d'ailleurs (de l'extérieur de la méthode) et peuvent se révéler être faux (donc il vaut mieux ne pas sortir en échec, et plutôt traiter le problème, avec des exceptions notamment),

— vérifier les résultats d'interaction avec un utilisateur (les interactions vont souvent comporter des erreurs, à traiter),

— assurer le déroulement correct du programme en présence d'un environnement défaillant.

Avec les assertions :

— il faut éviter de créer des effets de bord, c'est-à-dire de modifier l'état du programme (par exemple ne pas empiler ou dépiler dans une assertion).

Au contraire, en ce qui concerne les exceptions :

— elles peuvent contenir de l'information décrivant l'erreur et des méthodes de traitement au minimum palliatifs, gardant l'été du programme cohérent.

Enfin, il est normal que le code de gestion des contrats (assertions, exceptions) soit, dans de nombreux cas, plus important en dimension et en complexité que le code qui traite des cas d'exécution normaux du programme. Il y a souvent un effort important à faire pour la conception des contrats, l'écriture des assertions et la conception des hiérarchies d'exceptions associées au non respect des contrats. Par exemple, pour une classe `Pile`, on aura souvent une racine d'héritage `PileException` spécialisée par une hiérarchie de classes représentant toutes les erreurs pouvant se produire à l'exécution sur une pile. Pour toute écriture de classe et de programme, on devra se poser ces questions en parallèle de la conception des cas standards, afin de limiter au maximum les erreurs de fonctionnement et les arrêts brutaux des programmes avec perte possible de la cohérence des données associées.