TP de JAVA – Série 9
Tester et débugger

Université de Mons-Hainaut
2004 - 2005

Remarque : Il est en général conseillé d'utiliser les termes *déboguer*, *débogage*, *débogueur* et *bogue*. Cependant, dans la pratique et dans la langue orale nous constatons que ceux-ci sont peu utilisés. C'est pourquoi nous utiliserons les termes *débugger*, *débugage*, *débugueur* et *bug*.

1 Notions théoriques

1.1 Méthode de Héron

Si x est une estimation de $\sqrt{a}$, alors la moyenne de x et $\frac{a}{x}$ est une meilleur estimation de $\sqrt{a}$.

```
public class RootApproximator
{
   private double a;
   private double xold;
   private double xnew;

   /**
      Creates a RootApproximator object
      @param aNum is the input value
      (Precondition: aNumber >= 0)
   */
   public RootApproximator(double aNum)
   {
      a = aNum;
      xold = 1;
      xnew = a;
   }

   /**
      Compute a better guess from the current guess.
      @return the next guess
   */
   public double nextGuess()
   {
      xold = xnew;
      if(xold != 0)
         xnew = (xold + (a / xold)) / 2;
      return xnew;
   }

   /**
      Compute the root
      @return the computed value for the square root
   */
   public double getRoot()
   {
      while (!Numeric.approxEqual(xnew, xold))
         nextGuess();
      return xnew;
   }

}
```

1.2 Tests unité

Tester les classes, les méthodes individuellement avant de les incorporer dans un programme.

- Un *test unité* sert à tester une méthode ou un ensemble de méthodes coopératives.
- Pour un test unité, les classes contenant les méthodes à tester sont compilées hors du programme principal, avec une simple méthode (dans une classe de test) appelée *test harness* qui va faire appel et passer des paramètres aux méthodes à tester.

Exemple 1 :

```
import javax.swing.JOptionPane;

public class RootApproximatorTest
{
   /**
    Test harness for method nextGuess
   */
   public static void main(String[] args)
   {
      String input
           = JOptionPane.showInputDialog("Enter a number");
      double a = Double.parseDouble(input);
      RootApproximator r = new RootApproximator(a);

      for (int tries = 1; tries <= 10; tries++)
         System.out.println("Guess #"
              + tries + ": " + r.nextGuess());

      System.exit(0);
   }
}
```

Exemple 2 :

```
/**
  Test harness for method getRoot
*/
public static void main(String[] args)
{
   boolean done = false;
   while (!done)
   {
      String input = JOptionPane.showInputDialog(
          "Enter a number, cancel to quit");
      if (input == null)
         done = true;
      else
      {
         double a = Double.parseDouble(input);
         RootApproximator r = new RootApproximator(a);

         System.out.println("Square root of "
             + x + " = " + r.getRoot());
      }
   }
   System.exit(0);
}
```

1.3 Générer des entrées pour les tests

Problème : les tests précédents sont lents car les données doivent être rentrées à la main.

Solutions : lire les données à la console, dans un fichier ou générer des données aléatoirement.

Trois sortes de valeurs à tester (*test de cas*) :

- *tests positifs* : valeurs légitimes qui servent à vérifier si les résultats sont ceux attendus (e.g. pour $\sqrt{}$: 100, $\frac{1}{4}$, 0.01, 10^{12}, $\ldots$)
- *tests de bornes* : valeurs (légitimes) qui se trouvent à la limite des données acceptables (e.g. 0 pour $\sqrt{}$)
- *tests négatifs* : valeurs qui doivent être rejetées par le programme (e.g. -2 pour $\sqrt{}$)

1.4 Evaluer les résultats des tests

- utiliser des données dont on connaît les résultats
- vérifier les résultats à la main (lent)
- vérifier que les résultats respectent une propriété connue (e.g. si x est supposé être la racine carrée de a, alors x^2 devrait être égal à a.)
- utiliser un *oracle* (manière moins rapide de calculer le même résultat mais dont on est sûr de la validité). Exemple: calculer $x^{\frac{1}{2}}$ en utilisant `Math.Pow`.

Les deux dernières solutions ont l'avantage d'être automatiques (inclues dans le test harness).

Exemple 3 :

```
/**
  Test harness for getRoot using a known property
*/
public static void main(String[] args)
{
   int fail = 0;
   final int TRIES = 1000;
   Random generator = new Random();
   for (int i = 1; i <= TRIES; i++)
   {
      double x = 1.0E6 * generator.nextDouble();
      RootApproximator r = new RootApproximator(x);
      double y = r.getRoot();

      if (!Numeric.approxEqual(y*y,x))
         fail++;
    }
    System.out.println("Fail: " + fail + " on " + TRIES + " tries");
}
```

1.5 Collecter les tests

- Utiliser des fichiers réutilisables pour refaire les tests dans le futur.
- Une *batterie de tests* (anglais: *test suite*) est une collection de tests réutilisables.
- Phénomène de *cyclage* : un bug qui a été fixé peut réapparaître dans une version future : utile de garder un test pour chaque bug corrigé.
- *tests de régression* : garder tous les anciens tests et les tester à chaque nouvelle version du programme pour s'assurer que des défauts anciens ne réapparaissent pas.
- *black-box testing* : méthode de test qui ne tient pas compte de la structure interne d'un programme. Elle vérifie simplement que les tests positifs soient valides et que les tests négatifs soient rejettés correctement.
- *white-box testing* : tient compte de la structure interne du programme (e.g. tests unité pour chaque méthode).
- *test coverage* : mesure quel pourcentage du code est couvert par les tests (idéalement 100%, ce qui implique par exemple que tous les cas d'un `if/else` soient couverts).

1.6 Traçage de programme

Le fait d'ajouter des messages destinés aux débugages dans le code d'un programme permet le *traçage du programme*.

Problème : les classes finales doivent souvent être muettes. Le traçage des programmes peut mener à des affichages intempestifs si on en les efface pas.

Une solution : utiliser la classe `Logger` : voir livre et API

1.7 Utilisation d'un débugueur

Un débugueur est un programme qui permet d'éxécuter un autre programme *pas à pas* en y incorporant des points d'arrêts (*breakpoints*). Il permet d'inspecter les variables en cours d'éxécution.

1.8 Plan de l'HOWTO 4 : Débugger

Etape 1 *Reproduire l'erreur*

Etape 2 *Simplifier l'erreur*

Etape 3 *Diviser pour règner*

Etape 4 *Savoir ce que votre programme doit faire*

Etape 5 *Regarder les détails*

Etape 6 *Soyez sûrs de comprendre le bug avant de le corriger*

2 Exercices de révision

R9.1 : Définissez les notions de *test unité* et de *test harness*.

R9.2 : Qu'est-ce qu'un *oracle* ?

R9.3 : Définissez les notions de *tests de régression* et de *batterie de tests*.

R9.4 : Qu'est-ce que le *cyclage* ? Comment l'éviter ?

R9.5 : La fonction *arc sinus* est l'inverse de la fonction *sinus*, i.e., $y = \arcsin(x)$ si $x = \sin(y)$. Cette fonction n'est définie que si $-1 \leq x \leq 1$. Supposez que vous devez écrire une méthode Java pour calculer l'arc sinus. Donnez trois tests *positifs* et un test *de borne* avec les valeurs de retour supposées, ainsi que deux tests *négatifs*.

R9.6 : Qu'est-ce que le *tracage d'un programme* ?

R9.7 : Vrai ou faux :

- Si un programme passe avec succès tous les tests d'une batterie de tests, il ne contient plus de bugs.
- Si on *démontre* que toutes les méthodes d'un programme sont correctes, alors le programme ne contient pas de bugs.

3 Exercices de programmation

Rappel : La fonction *arc sinus* est l'inverse de la fonction sinus :

$$y = \arcsin(x) \text{ si } x = \sin(y),$$

où x est comris entre -1 et 1 et y est exprimé en radians. Par exemple,

$$\begin{aligned} \arcsin(0) &= 0, \\ \arcsin(1/2) &= \Pi/6, \\ \arcsin(\sqrt{2}/2) &= \Pi/4, \\ \arcsin(\sqrt{3}/2) &= \Pi/3, \\ \arcsin(1) &= \Pi/2, \\ \arcsin(-1) &= \Pi/2. \end{aligned}$$

P9.1 : Ecrivez une classe `ArcSinApproximator` qui calculera la fonction arc sinus à partir de sa série de Taylor :

$$\arcsin(x) = x + \frac{x^3}{3!} + \frac{3^2 \cdot x^5}{5!} + \frac{3^2 \cdot 5^2 \cdot x^7}{7!} + \frac{3^2 \cdot 5^2 \cdot 7^2 \cdot x^9}{9!} + \cdots$$

Remarques :

- Il existe une méthode Java dans la libraire standard pour calculer cette fonction mais vous ne devez pas l'utiliser pour cet exercice.
- Calculez la somme des termes jusqu'à ce qu'un nouveau terme soit plus petit que 10^{-6}.
- Ne calculez pas les puissances et les factorielles explicitement : calculez plutôt chaque terme à partir du précédant.
- Cette méthode sera réutilisée dans les exercices qui suivent.

P9.2 : Ecrivez un *test harness* pour la classe `ArcSinApproximator` qui lit des nombres réels et qui calcule leurs arc sinus. Vérifiez ensuite les valeurs avec une calculatrice scientifique ou à partir des exemples donnés ci-dessus.

P9.3 : Ecrivez un *test harness* qui génère *automatiquement* des tests de cas pour la classe `ArcSinApproximator` en prenant toutes les valeurs comprises entre -1 et 1, par pas de 0.1.

P9.4 : Ecrivez un *test harness* qui génère *aléatoirement* des tests de cas pour la classe `ArcSinApproximator` en prenant 10 nombres réels aléatoires compris entre -1 et 1.

P9.5 : Ecrivez un *test harness* qui teste automatiquement la validité de la classe `ArcSinApproximator` en vérifiant que

```
Math.sin(new ArcSinApproximator(x).getArcSin())
```

est approximativement égal à `x`. Testez-le sur 100 nombres aléatoires (compris entre -1 et 1).

P9.6 : La fonction arc sinus peut être calculée à partir de la fonction *arc tangeante* :

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right).$$

Utilisez cette expression comme un *oracle* pour tester que votre méthode arc sinus fonctionne correctement. Testez votre méthode à partir de 100 nombres aléatoires.

P9.7 : Le domaine de la fonction arc sinus est $-1 \leq x \leq 1$. Testez votre classe en calculant $\arcsin(1.1)$. Que se passe-t-il ?

P9.8 : Ajoutez des messages dans la boucle de la méthode qui calcule l'arc sinus par addition de termes successifs. Affichez l'exposant du terme courant, la valeur du terme courant, et l'approximation courante du résultat. Quelle trace de votre programme obtenez-vous en calculant $\arcsin(0.5)$?

P9.9 : Téléchargez les fichiers `RootApproximator.java` et `Numeric.java` sur le site. La classe `RootApproximator` contient deux bugs. Créez une série de tests de cas pour mettre à jour les bugs. Essayez ensuite de corriger les bugs en utilisant la technique du traçage du programme.