IN201 : 2 – Tests et associations

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Solution

Résumé

Le but de ce TP est de construire des tests unitaires avec JUnit et d'implanter une association en Java.

1 Contenu

Ce corrigé succinct contient les réponses au TP numéro 2. Le corrigé de la partie implantation en Java est disponible sur http://www.tofgarion.net/lectures/IN201.

2 Élémentaire mon cher Watson !

Vous avez fait l'erreur de laisser le soin à votre binôme de finir le premier TP d'IN201 dans lequel il fallait construire une classe `Orbite`. Malheureusement, il a préféré copier le code de la classe `Orbite` sur un autre binôme. Un malheur ne venant jamais seul, ~~cet idiot~~ il a voulu ajouter quelques « personnalisations » dans le code pour qu'on ne détecte pas le plagiat et « corriger » quelques erreurs et évidemment, la classe ne fonctionne plus. Pour couronner le tout, il est parti en soirée et ne vous a laissé que le *bytecode* de la classe `Orbite`, ayant effacé par mégarde le code source en le prenant pour un virus récupéré sur Internet...
Le but de cet exercice est de trouver les erreurs qui ont été introduites dans le code de la classe en utilisant des tests JUnit (et seulement des tests JUnit !). La classe boguée s'appelle `fr.isae.orbit.FalseOrbite`. Vous créerez donc une classe de test JUnit `fr.isae.orbit.FalseOrbiteTest` dans le répertoire `tests` de votre TP et écrirez les tests unitaires permettant de détecter les erreurs.
Voici quelques indications :

- il y a trois erreurs qui ont été introduites ;
- la classe `FalseOrbite` compile correctement ;
- il n'a pas touché à ce qui concerne le foyer ;
- il a pu enlever des bouts de code qu'il ne comprenait pas ;
- les erreurs peuvent être trouvées de façon indépendante en utilisant les tests unitaires appropriés (ils sont simples !) ;
- votre binôme n'a ni modifié les méthodes permettant de calculer la position d'un point, ni la méthode permettant de vérifier la visibilité.

D'un point de vue de l'implantation :

- la classe `Point` appartient maintenant au paquetage `fr.isae.geometry` ;
- votre classe de test devra appartenir au paquetage `fr.isae.orbit` ;
- lorsque vous avez trouvé une erreur, vous pouvez éventuellement la corriger dans les tests suivants en appelant les méthodes adéquates.

1. préparer une classe de test JUnit via Eclipse. En particulier, réfléchir aux acteurs nécessaires pour les tests.

   Solution :
   Rien de bien difficile si on a suivi le cours, Eclipse mâche le travail. Je propose d'utiliser deux acteurs pour les tests, l'orbite GPS et l'orbite EXOSAT. La classe de test JUnit est présentée sur le listing 1. Remarquez bien qu'il n'y a pas de constructeur dans la classe !

Listing 1– La classe `FalseOrbitTest`

```
package fr.isae.orbit;

import fr.isae.geometry.Point;

import org.junit.*;
import static org.junit.Assert.*;

/**
 * Unit Test for class FalseOrbite.
 *
```

```
 * Created: Tue 17 16:05:42 2013
 *
 * @author <a href="mailto:christophe.garion@isae.fr">Christophe Garion</a>
 * @version 1.0
 */
public class FalseOrbiteTest {

    private FalseOrbite gps;
    private FalseOrbite exosat;

    private static final double EPS = 1E-6;
    private static final double STEP = 1E-5;

    /**
     * Setup for the tests.
     */
    @Before public void setUp() {
        this.gps = new FalseOrbite(0.0, 26E6);
        this.exosat = new FalseOrbite(0.93, 100E6);
    }

    /**
     * Cleanup for the tests.
     */
    @After public void tearDown() {

    }
}
```

2. trouver les erreurs introduites dans la classe `FalseOrbite`.

Solution :

Vous remarquerez que j'ai utilisé dans ma classe de tests des *oracles de tests*, i.e. des instances de `Orbite` pour GPS et EXOSAT, car je sais que la classe `Orbite` est correcte. Attention, ce ne sont pas réellement des tests unitaires, car dans un test unitaire on ne doit utiliser que des valeurs constantes, alors que je fais des calculs pour trouver la valeur attendue dans mes tests.

Les trois erreurs étaient les suivantes :

1. la valeurs du demi-petit axe n'est pas la bonne, la méthode `calculerB` (cf. listing 2) étant fausse (votre binôme a en fait affecté la valeur de a à b !).

Listing 2– Méthode `calculerB`

```
312     private void calculerB() {
313         this.b = this.a * Math.sqrt(1.0 - this.e * this.e) /
314             Math.sqrt(1.0 - this.e * this.e);
315     }
```

Cette erreur pouvait être difficile à détecter : on ne peut pas tester directement la méthode `calculerB` car elle est privée[1]. Évidemment, des tests sur l'orbite GPS ne servaient à rien pour détecter cette erreur, vu qu'elle est circulaire. Un test basique de distance aux deux foyers de l'ellipse pour EXOSAT (cf. listing 3) pouvait mettre la puce à l'oreille, la valeur n'étant pas celle attendue. Vous remarquerez que j'utilise une méthode privée pour éviter de répéter le même code dans le test de distance bifocale pour GPS et EXOSAT. Il suffisait alors de tester

l'accesseur getB (cf. listing 4) pour se rendre compte de l'erreur. Après correction via setB, le test de la distance bifocale fonctionne (cf. listing 5).

Listing 3– Méthodes de test utilisant la distance bifocale

```
55  @Test public void testDistanceBifocaleEXOSAT() {
56      this.testDistanceFalseOrbite(this.fexosat);
57  }
58
59  private void testDistanceFalseOrbite(FalseOrbite o) {
60      double c = o.getC();
61
62      double distance = 2 * (o.getA() - c) + 2 * c;
63
64      Point foyer1 = o.getFoyer();
65      Point foyer2 = new Point(foyer1.getX() - 2 * c, foyer1.getY());
66
67      Point p = null;
68
69      for (double theta = 0; theta < 2 * Math.PI; theta += STEP) {
70          p = o.calculerPointSurOrbite(theta);
71          assertEquals(distance, p.distance(foyer1) + p.distance(foyer2), EPS);
72      }
73  }
```

Listing 4– Méthodes de test des accesseurs sur a et b

```
75  /**
76   * Test method for getA for EXOSAT.
77   */
78  @Test public void testGetAEXOSAT() {
79      assertEquals(this.exosat.getA(), this.fexosat.getA(), EPS);
80  }
81
82  /**
83   * Test method for getB for EXOSAT.
84   */
85  @Test public void testGetBEXOSAT() {
86      assertEquals(this.exosat.getB(), this.fexosat.getB(), EPS);
87  }
```

Listing 5– Méthodes de test avec correction

```
89  /**
90   * Test method for getB for EXOSAT with correction.
91   */
92  @Test public void testGetBEXOSATCorrect() {
93      this.fexosat.setB(this.exosat.getB());
94      assertEquals(this.exosat.getB(),
95                   this.fexosat.getB(), EPS);
96  }
97
98  /**
99   * Testing if the sum of the distances to the focus is constant
100  * for EXOSAT with correction.
101  */
102 @Test public void testDistanceBifocaleEXOSATCorrect() {
103     this.fexosat.setB(this.exosat.getB());
104     this.testDistanceFalseOrbite(this.fexosat);
```

```
105     }
```

2. on ne se préoccupait plus du sens du vecteur tangent à l'ellipse, alors que l'on souhaitait que celui-ci soit orienté dans la direction du mouvement du satellite (cf. ligne 268 du listing 6). Cette erreur se repérait facilement en utilisant un test basique, par exemple à $\frac{\pi}{2}$ sur l'orbite GPS (cf. listing 7). Il faut écrire d'autres tests pour être sûr de l'erreur.

Listing 6– Méthode calculerVecteurTangent

```
245     public Point calculerVecteurTangent(double theta) {
246         // on s'occupe des cas aux limites
247         if (Math.abs(theta % (2.0 * Math.PI)) < EPS) {
248             return new Point(0.0, 1.0);
249         }
250 
251         if (Math.abs(theta % Math.PI) < EPS) {
252             return new Point(0.0, -1.0);
253         }
254 
255         // calcul dans les autres cas
256         Point centre = new Point(this.foyer.getX() - this.getC(),
257                                  this.foyer.getY());
258 
259         Point p = this.calculerPointSurOrbite(theta);
260 
261         // on se ramene a un centre a l'origine
262         p.translater(-centre.getX(), -centre.getY());
263 
264         // on choisit un vecteur avec une abscisse qui vaut celle du
265         // demi-grand axe. Comme on sait que theta via atan2 sera
266         // compris entre -PI et PI, c'est facile...
267         double x = this.a + p.getX();
268 
269         double y = (1.0 - (x * p.getX()) / (this.a * this.a)) *
270             (this.b * this.b) / p.getY();
271 
272         // on normalise le vecteur
273         Point vec = new Point(x - p.getX(), y - p.getY());
274 
275         double lvec = vec.distance(Point.ORIGINE);
276 
277         return new Point(vec.getX() / lvec, vec.getY() / lvec);
278     }
```

Listing 7– Méthode de test détectant l'erreur sur le vecteur tangent

```
228     /**
229      * Test if tangent vector a PI/2 and -PI/2 is in the right
230      * direction.
231      */
232     @Test public void testTangentVector() {
233         Point veca = this.fgps.calculerVecteurTangent(Math.PI / 2.0);
234         Point vecb = this.fgps.calculerVecteurTangent(-Math.PI / 2.0);
235 
236         assertEquals(-1.0, veca.getX(), EPS);
237         assertEquals(0.0, veca.getY(), EPS);
238         assertEquals(1.0, vecb.getX(), EPS);
239         assertEquals(0.0, vecb.getY(), EPS);
240     }
```

3. le calcul de θ à partir de v était faux, car on calculait l'angle au centre au lieu de l'anomalie excentrique (cf. listing 8). Cette erreur pouvait être difficile à détecter, car encore une fois on ne peut pas tester directement la méthode privée `calculerTheta`. Par contre, on pouvait trouver l'erreur en utilisant une position caractéristique de l'ellipse, le *latus rectum*, point situé à la verticale du foyer et pour lequel la distance au foyer est connue et vaut $\frac{b^2}{a}$ (cf. listing 9). Vous remarquerez que j'ai utilisé les bonnes valeurs de a et b pour le test.

Listing 8– Méthode `calculerTheta`

```
317     private double calculerTheta(double v) {
318         double r = this.a * (1.0 - this.e * this.e) /
319             (1.0 + this.e * Math.cos(v));
320 
321         double opp = Math.sin(v) * r;
322         double adj = Math.cos(v) * r;
323 
324         return Math.atan2(opp, this.getC() + adj);
325     }
```

Listing 9– Méthode de test détectant l'erreur sur le calcul de θ

```
242     /**
243      * We know that when v = PI/2, y is the latus rectum and is
244      * b^2/a.
245      */
246     @Test public void testLatusRectumEXOSAT() {
247         Point lr = this.fexosat.calculerPointSurOrbiteFoyer(Math.PI / 2.0);
248 
249         assertEquals(2.0 * Math.pow(this.fexosat.getB(), 2) / this.fexosat.getA(),
250                     lr.getY() - this.fexosat.getFoyer().getY(), EPS);
251     }
```

Je ne montre ici que des tests minimaux, il faut en faire plus à mon avis pour trouver les erreurs. Vous trouverez sur le site une classe de test complète pour `Orbite` qui vous donnera quelques indications sur les tests que l'on peut faire.

3 Une orbite, c'est aussi des points...

On décide d'implanter maintenant la classe `OrbiteDiscrete` définie en cours. Le diagramme d'analyse de la classe est présenté sur la figure 1 et un diagramme d'implantation de la classe `OrbiteDiscrete` est présenté sur la figure 2.
Le mot-clé {ordered} situé sur l'association entre `OrbiteDiscrete` et `Point` est ce que l'on appelle une *contrainte* : elle nous indique les points sont stockés suivant un certain ordre. On ne peut donc pas utiliser un ensemble pour les stocker, on utilisera une liste lors de l'implantation.
Quelques remarques sur le diagramme d'implantation :

- l'objet de type `Orbite` passé en paramètre du constructeur n'est pas stocké comme attribut, il sert juste à construire les points ;
- le **double** passé en paramètre du constructeur sert de pas pour faire varier l'angle par rapport au foyer lors de la construction des points ;
- le calcul du vecteur tangent utilise les points précédant et suivant le point considéré ;
- l'homothetie se fait par rapport au centre de l'ellipse.

1. créer la classe `OrbiteDiscrete` dans Eclipse, préparer le squelette de la classe (attributs, méthodes) et la documenter.
2. écrire le constructeur de la classe `OrbiteDiscrete`.

Solution :
Rien de bien difficile, le constructeur est présenté sur le listing 10.

Listing 10– Le constructeur de `OrbiteDiscrete`

```
21     /**
```

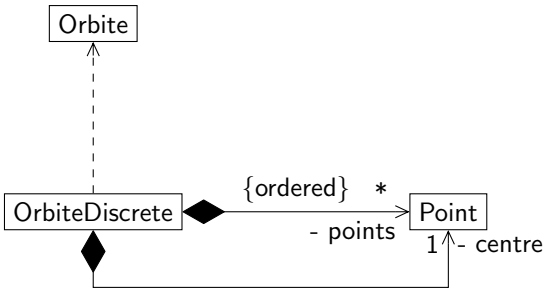

FIGURE 1 – Diagramme de classe d'analyse de `OrbiteDiscrete`

fr::isae::orbit::OrbiteDiscrete
- points: java::util::ArrayList<fr::isae::geometry::Point> - centre: fr::isae::geometry::Point
+ OrbiteDiscrete(o: Orbite, pasAngle: double) - OrbiteDiscrete(points: java::util::ArrayList<fr::isae::geometry::Point>, centre: fr::isae::geometry::Point) + getNbPoints(): int + getPoint(i: int): fr::isae::geometry::Point + getCentre(): fr::isae::geometry::Point + getCirconference(): double + getVecteurTangent(i: int): Point + translater(dx: double, dy: double) + homothetie(rapport: double) + clone(): OrbiteDiscrete + equals(o: OrbiteDiscrete): boolean + toString(): String

FIGURE 2 – Diagramme de classe d'implantation de `OrbiteDiscrete`

```
22      * Creer une instance de <code>OrbiteDiscrete</code> en utilisant
23      * une orbite definie geometriquement.
24      *
25      * @param orbite l'orbite geometrique servant a calculer la
26      *               sequence de points
27      * @param pasAngle le pas servant a incrementer l'angle pour
28      *                 le calcul des points
29      */
30     public OrbiteDiscrete(Orbite orbite, double pasAngle) {
31         this.points = new ArrayList<Point>();
32
33         for (double angle = 0.0;
34              angle < 2.0 * Math.PI;
35              angle += pasAngle) {
36             this.points.add(orbite.calculerPointSurOrbite(angle));
37         }
38
39         this.centre = orbite.getFoyer();
40         this.centre.translater(-orbite.getC(), 0.0);
41     }
```

3. en considérant le diagramme d'analyse présenté sur la figure 1 et le principe d'encapsulation, quel est l'impact sur la méthode `getPoint` ?

Solution :

La visibilité du rôle accordé aux points constituant une instance de `OrbiteDiscrete` est privée. On ne doit donc pas accéder directement aux points depuis l'extérieur. Si on utilise simplement **`return this`**`.point.get(i)` dans `getPoint`, un utilisateur extérieur récupérera une référence vers le point demandé et pourra donc le modifier directement (ce qui viole également le principe d'encapsulation). Il fallait donc utiliser la méthode `clone` de `Point` pour renvoyer le point (cf. listing 11). Vous remarquerez que le même principe est appliqué pour `getCentre`. J'ai choisi ici de commencer l'indexation à 1, il faut faire attention car l'indexation pour `ArrayList` commence à 0.

Listing 11– La méthode `getPoint`

```
62  /**
63   * Retourner un point de l'orbite.
64   *
65   * @param i un index, compris entre <code>1</code> et
66   *          <code>this.getNbPoints()</code>
67   * @return le ieme point dans la sequence
68   */
69  public Point getPoint(int i) {
70      return this.points.get(i - 1).clone();
71  }
```

4. implanter la méthode `equals` de `OrbiteDiscrete`. On considérera que deux instances de `OrbiteDiscrete` sont égales si elles ont le même nombre de points, si tous leurs points ont des coordonnées identiques et si les centre des deux instances ont des coordonnées identiques.

Solution :

Rien de particulier, la méthode est présentée sur le listing 12.

Listing 12– La méthode `equals`

```
174 /**
175  * Verifier si deux orbites sont egales. Pour cela on verifie qu'elles
176  * ont le meme nombre de points, que tous leurs points sont egaux (au
177  * sens de <code>equals</code>) et que leurs centres sont egaux.
178  *
179  * @param o l'orbite dont on veut verifier si elle est egale a
180  *          <code>this</code>
181  * @return un booleen qui est <code>true</code> si les deux orbites
182  *         sont egales
183  */
184 public boolean equals(OrbiteDiscrete o) {
185     if (o == null) {
186         return false;
187     }
188
189     if (this.points.size() != o.points.size()) {
190         return false;
191     }
192
193     if (!this.centre.equals(o.centre)) {
194         return false;
195     }
196
197     for (int i = 0; i < this.points.size(); i++) {
198         if (!this.points.get(i).equals(o.points.get(i))) {
199             return false;
200         }
201     }
202
```

```
203         return true;
204     }
```

5. en considérant le diagramme d'analyse présenté sur la figure 1, quel est l'impact sur la méthode `clone` ? À quoi sert le constructeur privé prenant en paramètre une `ArrayList` ?

   Solution :

   La méthode `clone` doit servir à construire une nouvelle instance de `OrbiteDiscrete` à partir de l'instance courante. Il faut que cette nouvelle instance soit égale à l'instance courante au sens de la méthode `equals`. Si vous regardez l'implantation de la méthode `equals`, deux instances de `OrbiteDiscrete` seront égales si elles le même nombre de points, des centres avec des coordonnées identiques et si tous leurs points ont des coordonnées identiques. De plus, le diagramme d'analyse nous précise que la relation entre `OrbiteDiscrete` et `Point` est une *composition*. Un point ne peut donc appartenir qu'à une seule orbite. Pour cloner une instance de `OrbiteDiscrete`, il faut donc :

   - créer une nouvelle instance d'`ArrayList` ;
   - cloner les points présent dans l'instance courante de `OrbiteDiscrete` et les ajouter à l'instance d'`ArrayList` nouvellement créée ;
   - créer une nouvelle instance de `OrbiteDiscrete` en utilisant la nouvelle instance de `ArrayList` et un clone du centre de l'instance courante de `OrbiteDiscrete`, car le principe d'encapsulation nous interdit de partager une instance de `Point` représentant le centre entre deux orbites.

   Il faut donc un constructeur prenant en paramètre une instance de `ArrayList<Point>` et une instance de `Point` représentant le centre. Ce constructeur doit être privé et n'être utilisé que par `clone`, car il sera difficile de vérifier que la liste de points utilisée dans ce constructeur par un utilisateur extérieur représente bien une ellipse.

   Attention, la méthode `clone` de `ArrayList` réalise ce que l'on appelle une *shallow copy* : les points de la liste ne seront pas clonés ! Si on utilise `clone` sur une liste, on obtiendra bien une nouvelle liste, mais contenant des références vers les points de la liste initiale. Il fallait donc faire la copie « à la main ».

   Le code source de la méthode `clone` est présenté sur le listing 13.

Listing 13– La méthode `clone`

```
206     /**
207      * Cloner une orbite. L'orbite obtenue est un objet different, mais est
208      * egale a <code>this</code>.
209      *
210      * @return un clone de <code>this</code>
211      */
212     public OrbiteDiscrete clone() {
213         return new OrbiteDiscrete(this.points, this.centre);
214     }
```

6. implanter le reste des méthodes de la classe `OrbiteDiscrete`.

   Solution :

   Rien de bien compliqué, le code source est disponible sur http://www.tofgarion.net/lectures/IN201. Il fallait profiter au maximum de la boucle **for** utilisable sur les collections pour faciliter l'écriture de la classe. Quelques oublis classiques :

   - translater le centre quand on translate les points ;
   - se ramener à un centre à l'origine avant de faire l'homothétie via la méthode `setModule` de `Point`.

   On se rend compte qu'un objet de type `OrbiteDiscrete` *délègue* beaucoup de ses services à d'autres classes. Par exemple, pour translater l'orbite, on utilise la méthode `translater` de `Point` sur chacun des poitns. Formellement, il faudrait utiliser la notion de *port* et de *connecteur de délégation* en UML 2.0 (voir [1] pour plus de détails). On peut toutefois utiliser les notes comme sur la figure 3 pour représenter cela.

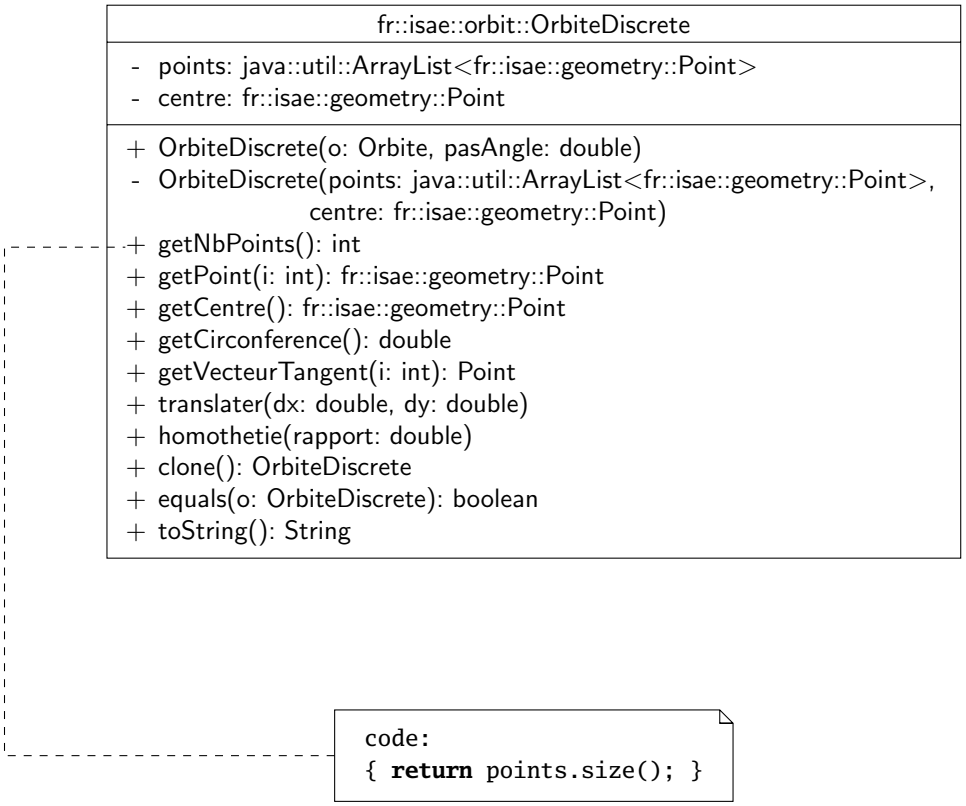

FIGURE 3 – Diagramme de classe d'implantation de `OrbiteDiscrete` avec note

4 Retour sur les tests unitaires de `OrbiteDiscrete`

Vous pourrez voir que j'ai effectué de nombreux tests unitaires sur la classe `OrbiteDiscrete`. Beaucoup d'entre eux sont inspirés des tests unitaires de la classe `Orbite` que vous trouverez sur le site. J'ai utilisé pour les tests 4 orbites :

- une orbite construite à partir de l'orbite GPS
- une orbite construite à partir de l'orbite GPS avec foyer décalé
- une orbite construite à partir de l'orbite EXOSAT
- une orbite construite à partir de l'orbite EXOSAT avec foyer décalé

Évidemment, mes tests ne sont pas exhaustifs, mais le fait d'utiliser 2 orbites très différentes et de décaler les foyers nous permet d'avoir une bonne confiance dans les tests. Vous remarquerez que pour éviter d'écrire 4 fois le même code de test, j'ai eu recours presque systématiquement à une méthode *privée* prenant en paramètre une instance de `OrbiteDiscrete` (et éventuellement d'autres paramètres comme une instance de `Orbite`) que j'appelle ensuite dans la méthode de test avec différentes orbites. Par exemple, le listing 14 présente la méthode testant `getCirconference` et la méthode privée associée (qui utilise une approximation de la circonférence d'une ellipse trouvée sur Wikipedia...).

Listing 14– La méthode testant `getCirconference`

```
102     /**
103      * Test orbit circonference using an approximation
104      * (see http://en.wikipedia.org/wiki/Ellipse#Equations).
105      */
106     @Test public void testOrbitCirconference() {
107         testCirconference(this.gpsp, this.gps);
108         testCirconference(this.gpsfp, this.gpsf);
109         testCirconference(this.exosatp, this.exosat);
110         testCirconference(this.exosatfp, this.exosatf);
111     }
112
113     private void testCirconference(OrbiteDiscrete op, Orbite o) {
114         double a = o.getA();
```

```
115         double b = o.getB();
116         double h = (Math.pow(a - b, 2)) / (Math.pow(a + b, 2));
117
118         double c = Math.PI * (a + b) * (1.0 + (3.0 * h) / (10 + Math.sqrt(4.0 - 3.0 * h)));
119
120         assertEquals(c, op.getCirconference(), o.getA() / 1E5);
121     }
```

On peut se demander si mes méthodes ont la bonne « granularité ». En effet, si `testOrbitCirconference` échoue, il va être difficile de savoir pour quelle orbite le test a échoué. Il aurait mieux valu créer une méthode de test pour chaque orbite, toujours en utilisant la méthode privée, mais j'ai été paresseux. . .
Vous remarquerez que pour tester `equals` je n'ai pas utilisé directement `assertEquals` avec deux instances de `OrbiteDiscrete` mais que je suis passé par la méthode `assertTrue` via `assertTrue(o.equals(o2))`. Nous reparlerons de cela lors du cours sur l'héritage.
J'ai enfin utilisé une fonctionnalité **avancée** de JUnit, les théories [2]. Cette utilisation correspond aux imports supplémentaires par rapport à une classe de test JUnit « classique », à l'utilisation de l'annotation **`@RunWith`** avant la déclaration de la classe et aux déclarations des attributs statiques préfixés par l'annotation **`@DataPoint`**. Je voulais vérifier que les orbites étaient différentes deux à deux au sens de `equals`. J'aurais donc dû écrire 12 assertions, ce qui est fastidieux. Les théories JUnit permettent de générer combinatoirement un ensemble de tests en utilisant des jeux de données représentés par les attributs statiques préfixés par **`@DataPoint`**. La méthode `testNotEquals` (cf. listing 15) utilise donc les 4 orbites définies pour générer tous les cas de tests possibles. La méthode `assumeTrue` permet de filtrer les données et donc d'éviter de comparer une orbite à elle-même.

Listing 15– La méthode `testNotEquals`

```
141     /**
142      * Test all possible combinations for non-equals orbits.
143      *
144      * @param o1 an orbit to use for combinations
145      * @param o2 an orbit to use for combinations
146      */
147     @Theory public void testNotEquals(OrbiteDiscrete o1, OrbiteDiscrete o2) {
148         assumeTrue(o1 != o2);
149
150         assertFalse(o1.equals(o2));
151     }
```

Vous n'avez bien sûr pas à utiliser de telles fonctionnalités, mais cela peut vous servir plus tard (ou si vous êtes curieux).

Références

[1] G. Booch, J. Rumbaugh et I. Jacobson. *The Unified Modeling Language reference manual*. Addison-Wesley, 2004.

[2] JUnit Team. *JUnit*. 2013. URL : http://www.junit.org.

License CC BY-NC-SA 3.0