IN201 : 2 – Tests et associations

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

Résumé

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

1 Objectifs

Les objectifs de ce TP sont les suivants :
- commencer à utiliser l'environnement de développement Eclipse ;
- tester une classe dont on ne possède pas le code avec JUnit ; `Orbite` ;
- implanter une association entre classes en utilisant une collection Java.

2 Utilisation d'Eclipse

Nous allons à partir de ce TP utiliser Eclipse [1], un environnement de développement logiciel complet. Pour pouvoir l'utiliser, suivez les indications disponibles sur http://www.tofgarion.net/lectures/IN201. On construit dans le tutoriel proposé une classe `Segment` en utilisant la classe `Point`.

3 Dépôt Subversion

L'URL de votre dépôt est pour cette semaine https://eduforge.isae.fr/repos/IN201/0921/GPE/bNUM où :
- GPE est le nom de votre groupe de PC en majuscule
- NUM est votre numéro de binôme (votre PC(wo)man les possède)

Par exemple, l'URL du dépôt du binôme 9 du groupe S est https://eduforge.isae.fr/repos/IN201/0921/S/b9. Pour connaître votre numéro de binôme, connectez-vous sur http://www.tofgarion.net/lectures/IN201.

Pour réaliser le TP, vous allez devoir créer un projet Java sous Eclipse en utilisant votre dépôt Subversion. Si vous avez configuré votre dépôt pour qu'il soit disponible dans la vue *SVN Repository Exploring* d'Eclipse, vous créérez votre projet en faisant un *checkout* **à partir du répertoire de votre dépôt** en cliquant droit dessus. N'oubliez pas de configurer votre *build path* pour pouvoir utiliser les éventuelles archives JAR placées dans le répertoire `lib`.

4 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.
2. trouver les erreurs introduites dans la classe `FalseOrbite`.

Vous ne pouvez pas utiliser `assertEquals` avec deux instances de `Point`, même si `Point` possède une méthode `equals`. Il faut utiliser `assertTrue` :

```
assertTrue(p1.equals(p2));
```

Nous reviendrons sur ce point lors du cours sur l'héritage.

N'oubliez pas de *commiter* le code source de votre classe.

5 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.

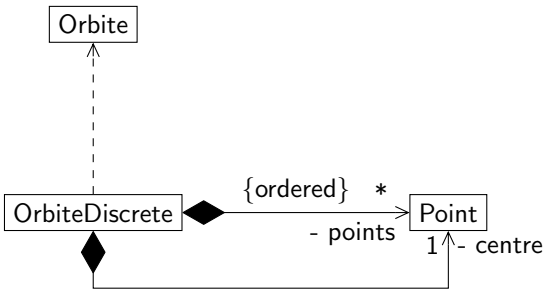

FIGURE 1 – Diagramme de classe d'analyse de `OrbiteDiscrete`

Vous implanterez les méthodes de la classe `OrbiteDiscrete` au fur et à mesure et vous créerez des méthodes de tests JUnit pour vérifier à chaque implantation qu'il n'y a pas d'erreurs.

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`

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

N'oubliez pas de *commiter* le code source de vos classes.

Références

[1] THE ECLIPSE FOUNDATION. *The Eclipse Foundation open source community website*. 2013. URL : http://www.eclipse.org/.

License CC BY-NC-SA 3.0