TP6 : Correction de programme avec Eclipse

TOP: Techniques and tOols for Programming

Première année

Préliminaires sur l'usage d'eclipse à l'ESIAL

Comme vous avez pu voir lors de la première séance utilisant Eclipse, le programme est un peu capricieux tel qu'il est configuré par défaut à l'ESIAL. Pour améliorer cela, il faut utiliser la JVM de Sun au lieu de celle par défaut. Pour cela, démarrez l'environnement avec la commande suivante sous Linux : `eclipse -vm /usr/lib/jvm/java-6-sun/bin/java`. Sous windows, donnez de manière similaire le chemin complet vers la machine virtuelle sun au démarrage d'eclipse depuis la ligne de commande.

★ **Exercice 1. JUnit et Test Driven Development** (d'après Christian Doerr).

Les tests unitaires sont à la base des méthodologies de programmation modernes comme l'eXtrem Programming (XP). Chaque test évaluant un aspect du programme, ils tendent à être courts et simples. Cette approche mène à écrire de nombreux tests. Pour être efficaces, ces tests doivent être exécutés souvent, par exemple après chaque changement. De cette façon, le développeur peut détecter immédiatement si son changement casse une fonctionnalité préexistante du programme.

L'XP va plus loin en demandant aux programmeurs d'écrire les tests avant que le code. Cet aspect s'appelle «Test Driven Development». L'objectif de cet exercice est de mettre JUnit et le TDD en oeuvre au travers du développement d'une structure de données simple.

▷ **Question 1.** Créez un nouveau projet Eclipse nommé «IntegerStack»

▷ **Question 2.** Créez un nouveau fichier Java dans le projet, nommé «IntStack»

▷ **Question 3.** Ajoutez le code suivant à votre fichier «IntStack».

/home/depot/1A/TOP/IntStack.java

```
1  import java.util.List;
2  import java.util.LinkedList;
3
4  public class IntStack {
5    private List<Integer> values;
6    public IntStack() {
7      this.values = new LinkedList<Integer>();
8    }
9    public void push(int x) {
10     this.values.add(0, x);
11   }
12   public int pop() {
13     return this.values.remove(0);
14   }
15   public int peek() {
16     return this.values.get(0);
17   }
18 }
```

Comme vous pouvez le voir, nous allons implémenter la structure de pile (que vous verrez plus en détail dans le module sur les structures de données). Pour cela, nous allons utiliser une structure de liste chaînée fournie par le langage Java.

Les primitives sur la pile sont simplement `push` et `pop` qui permettent respectivement d'ajouter un élément au sommet de la pile et de retirer l'élément au sommet de la pile. `peek` renvoie l'élément du sommet sans modifier la pile. Les autres primitives que l'on pourrait imaginer, comme par exemple celle indiquant le nombre d'éléments dans la pile, n'existent pas en général pour les piles, et il n'est pas demandé de les écrire.

▷ **Question 4.** Ajouter un test unitaire à votre projet en utilisant le menu "Nouveau → JUnit test case". Indiquez Swing comme TestRunner de votre suite de tests.

Eclipse vous indique qu'il n'a pas trouvé junit.jar et vous demande si vous souhaitez l'installer. Répondez oui, et indiquez lui où le trouver (/home/depot/1A/TOP). Si vous ratez cette étape, vous pouvez l'ajouter par le menu "Propriété du projet → Chemin de génération → Bibliothèques → Ajouter un jar externe".

▷ **Question 5.** Ajoutez le code ci-contre à votre classe de test. Comme le nom de la méthode commence par `test`, JUnit sait qu'il s'agit d'un test. Nous avons vu en cours ce à quoi sert la méthode assertTrue() de la ligne 4.

```
1 public void testPopAfterPush() {
2   IntStack stack = new IntStack();
3   stack.push(10);
4   assertTrue(stack.pop() == 10);
5 }
```

▷ **Question 6.** Exécutez votre suite de test d'un clic droit sur la classe, puis en sélectionnant "Exécuter comme → JUnit Test". Cela exécutera tous vos tests, et vous informera d'éventuels échecs. Il ne devrait pas y en avoir avec notre test.

▷ **Question 7.** Ajoutez les trois tests suivants à votre suite :
- Créer une pile et faire un pop (sans push).
- Faire plusieurs push, et vérifier le résultat d'un pop.
- Faire plusieurs push, suivis d'autant de pop (dont vous vérifierez la valeur de retour).

▷ **Question 8.** Tous vos tests commencent par créer une pile. Cette action devrait donc être placée dans une méthode `setUp()`. Faites le changement, puis vérifiez que votre suite de test fonctionne encore.

▷ **Question 9.** Écrivez une méthode `lookup(x)` qui retourne vrai si la valeur x est dans la pile et faux si non. Appliquez l'approche XP pour cela : écrivez tout d'abord quelques tests visant à vérifier la méthode lookup avant de l'écrire, puis utilisez les échecs de la suite de test pour guider votre développement.

★ **Exercice 2. Debugging**. (d'après Christian Doerr)

L'objectif de cet exercice est d'utiliser le debugger intégré d'Eclipse pour corriger un programme fourni.

▷ **Question 1.** Créez un projet, nommé DebuggingLab. Ajoutez à votre projet le fichier SortingAlgorithm.java que vous trouverez dans le dépot sur neptune.

▷ **Question 2.** Exécutez ce programme (qui contient une fonction main pour tester sur le tableau {1,5,9,3,2}) Vous remarquez qu'il lance une ArrayOutOfBoundsException, c'est à dire que le code accède à un index en dehors des limites d'un tableau.

▷ **Question 3.** Lancez maintenant votre programme dans le debugger (clic droit sur le test, puis "Debug As → Java Application"). Cette fois, quand votre programme lève une exception, le debugger démarre automatiquement et vous montre l'état du programme en cours d'exécution au moment où l'exception est lancée.
Voici quelques unes des choses que l'on peut faire avec un debugger :

1. *Voir la ligne qui cause le problème.*
   Le panel du milieu vous montre le source du fichier SortingAlgorithm.java, et il met également en valeur la ligne que le programme était en train d'exécuter au moment du problème. Comme le programme est mort d'une exception, la ligne surlignée est celle qui a déclanché l'exception.
2. *Examiner les variables.*
   Le panel en haut à droite nommé "Variables" montre les valeurs de toutes les variables visibles depuis le point courant (dans le scope), ainsi que leur valeur actuelle. Remarquez que vous pouvez explorer la valeur des variables de type tableau (comme a) en cliquant sur le '+' près de leur nom.
3. *Examiner la pile d'appel*
   Le panel Debug en haut à gauche vous permet d'observer la pile d'appel (ie, qui a appellé qui) de l'instant où le bug s'est produit. Par exemple, la vue courante indique que nous sommes actuellement dans qSortHelper (le plus récent est listé en haut), qui a été appellé par qSort, qui à son tour a été appellé par main. Cliquez sur "qSort" ou "main" et le debugger vous montrera la ligne de cette méthode qui a fait l'appel. La vue "variables" est également mise à jour quand vous cliquez de la sorte.
4. *Poser un point d'arrêt et exécuter le code pas à pas.*
   Lorsque vous trouvez un point où le programe crashe, le mécanisme décrit ci-dessus vous permet de voir l'état des lieux au moment du crash. Cependant, il est souvent pratique d'explorer cet état quelques temps avant que le crash n'ait lieu. Il vous faut pour cela utiliser des points d'arrêts (*breakpoints*), qui sont des marqueurs demandant au debugger d'arrêter l'exécution du programme pour que vous puissiez l'inspecter.
   Supposons que vous posiez un point d'arrêt sur la ligne "while (up < down)" dans qSortHelper. Pour celà, activez la vue Java (cliquez sur la petite icône ressemblant à une fenêtre avec un signe

plus tout en haut à droite de votre fenêtre Eclipse). Faites un clic droit sur la barre grise à gauche du source, en face de la ligne où vous souhaitez placer le point d'arrêt. Choisissez "Toggle Breakpoint", qui va activer un point d'arrêt si aucun n'existait, et desactiver un éventuel point d'arrêt placé à cette ligne. Quand un point d'arrêt est placé, vous observez un petit rond bleu sur la barre grise. Relancez maintenant votre programme dans le debugger, et il sera interrompu dès qu'il atteindra le point d'arrêt.

Lorsque vous etes arrété à un point d'arrêt, le menu "Run" contient plusieurs commandes d'intéret. "Step Into" et "Step Over" exécutent une instruction de votre programme. Si cette instruction est un appel de méthode, la première "entre" dans la fonction appellée pour vous permettre de l'étudier tandis que la seconde "saute" la méthode appellée pour aller directement à l'instruction suivante après l'appel. Si l'instruction n'est pas un appel de méthode, les deux commandes sont indentiques. "Resume" relance le programme jusqu'au prochain point d'arrêt ou la prochaine erreur, ou jusqu'à la fin du programme.

Il est également possible de créer des points d'arrêts conditionnels, qui n'arrêtent le programme que si une condition particulière est remplie, mais il est peu probablement que vous en ayez besoin ici.

5. *Modifier les variables.*
   Il est également possible de modifier la valeur des variables de votre programme en cours d'exécution d'un clic droit sur votre variable à changer (dans le panel en haut à droite), puis en sélectionnant "Change Value".

▷ **Question 4. Chercher 5 problèmes.**

Le code fourni contient au moins 5 bugs qu'il vous faut maintenant découvrir, puis corriger. Vous devez appliquer une approche par tests unitaires. Il vous est donc demandé d'écrire une classe de tests, puis d'écrire des tests permettant de mettre en valeur les fautes présentes dans le code.

Il n'est pas toujours simple d'imaginer les scénarios de tests qui vont provoquer les erreurs dans un programme. Vous pouvez au besoin vous inspirer de la méthode décrite à l'adresse suivante :
`http://www.engr.uvic.ca/~seng271/pdf/06_right_bicep.pdf`

▷ **Question 5. Corriger les problèmes, et documenter.**

Chaque fois que vous trouvez un problème grâce à un test, vous devez également le corriger. Documentez dans la classe de test le problème qui était présent, ainsi que comment vous avez choisi de le corriger.