Environnements de développement

Examen

15 mai 2008

Durée : 3 heures. Tous les documents sont autorisés.

Avant de commencer : s'assurer que la machine virtuelle utilisée est celle de Sun et non la version GNU (dans *Window / Preferences.../ Java / Installed JREs*), et qu'Eclipse est configuré pour compiler du Java 5.0 (dans *Window / Preferences.../ Java / Compiler / JDK Compliance*).

Modalités de remise des copies : en plus de remettre une copie papier, envoyer par e-mail à l'adresse labatut@di.ens.fr les projets des deux problèmes sous forme d'archive compressée `zip` (sélectionner le projet ouvert à archiver dans le *Package Explorer* et choisir *File / Export.../ General / Archive File*).

Pour chaque question, décrire de manière succincte (quelques lignes maximum) comment Eclipse a été utilisé pour réaliser l'opération demandée.

Il faudra attacher une attention toute particulière à la présentation du code : indentation et (éventuels) commentaires.

1 Utilisation du JDT

L'objectif de ce problème est de définir deux classes qui collectent des informations concernant des fichiers sources (nombre de lignes et nombre de lignes commentées) et qui les trient d'après ces informations.

1.1 Projet

Exercice 1 : Créer un projet Java `SourceCodeInfo` et rajouter un *package* `sci` à ce projet.

Exercice 2 : Rajouter à ce *package* une classe `SourceCodeInfo` avec comme champs privés : une chaîne de caractères `name`, un entier `numberOfLines` et un entier `numberOfCommentLines`.

Exercice 3 : Rajouter un constructeur (à 3 paramètres) ainsi que des accesseurs et affecteurs pour chacun de ces champs.

Exercice 4 : Rajouter au *package* `sci` une classe `SourceCodeInfoList` qui implémente une liste d'objets de type `SourceCodeInfo` sous la forme d'un simple tableau d'`Object`, avec :

- un champ privé de type `Object[]`,
- un constructeur (sans paramètre) qui initialise le précédent tableau à une longueur nulle,
- les méthodes publiques suivantes :
  - `int size()` qui renvoie la taille de la liste,
  - `SourceCodeInfo get(int i)` qui renvoie l'élément en position `i` dans la liste,
  - `boolean add(SourceCodeInfo sci)` qui essaie de rajouter un nouvel élément à la liste : si un élément avec le même champ `name` existe déjà alors la méthode renverra `false`, sinon, le tableau interne sera modifié, le nouvel élément ajouté à la fin et la méthode renverra `true`,
  - `void clear()` qui réinitialise la liste.

On souhaite maintenant trier la liste précédente suivant les différents champs de ses éléments.

Exemple : si l'on trie la liste à 3 éléments suivante d'après le champs `name` :

```
{ { "Fichier3.java", 10, 10},
  { "Fichier1.java", 30, 20},
  { "Fichier2.java", 20,  5}}
```

on doit obtenir la nouvelle liste :

```
{ { "Fichier1.java", 30, 20},
  { "Fichier2.java", 20,  5},
  { "Fichier3.java", 10, 10}}
```

Exercice 5 : Comment chercher les différentes méthodes de tri (*sort* en anglais) disponibles dans le JRE depuis Eclipse ?

Parmi les méthodes de tri disponibles, on trouve dans la classe `java.util.Arrays` une méthode qui trie des tableaux d'`Object` d'après une relation d'ordre indiquée par un objet d'une classe implémentant l'interface `java.util.Comparator`.

Exercice 6 : Implémenter dans la classe `SourceCodeInfoList`, les méthodes publiques suivantes :
- `void sortByName()`,
- `void sortByNumberOfLines()` et
- `void sortByNumberOfCommentLines()`,

qui trient par ordre croissant la liste d'objets d'après leurs champs `name`, `numberOfLines` ou `numberOfCommentLines`.

Remarque : Comme la méthode `Arrays.sort()` requiert un objet de type `java.util.Comparator` pour trier les objets, il faudra créer une classe implémentant cette interface dans chacun des cas (en se contentant d'invoquer les méthodes `compareTo()` et `equals()` des champs à comparer).

1.2 Tests unitaires

Exercice 7 : Ajouter la bibliothèque JUnit au chemin d'accès aux classes du projet.
Exercice 8 : Rajouter un *package* `sci.test` au projet et dans ce *package*, ajouter une classe de tests unitaires `SourceCodeInfoListTest` qui définit des méthodes de test pour chacun des 3 tris précédents.
Exercice 9 : Rajouter à la classe de tests unitaires `SourceCodeInfoListTest` une méthode de test vérifiant que le comportement de la méthode `boolean add(SourceCodeInfo sci)` est bien conforme à ses spécifications.
Exercice 10 : Vérifier que la classe `SourceCodeInfoList` passe les tests unitaires définis.

1.3 Débogage

Exercice 11 : À l'aide du débogueur, inspecter à l'exécution le contenu d'un objet de la classe `SourceCodeInfoList` et identifier les différents champs de la classe `java.lang.String` (bien détailler la procédure).

2 Utilisation du PDE

L'objectif de ce deuxième problème est de réutiliser les classes précédentes afin d'implémenter un *plug-in* Eclipse fournissant une nouvelle vue : cette vue permettra de trier les fichiers sources des projets courants, d'après les informations qui y sont attachées.

Exercice 12 : Créer un projet de *plug-in* intitulé `SourceCodeInfoPlugIn` dans l'espace de travail. Quel est le *template* de *plug-in* à utiliser ? Quel est le nom exact du point d'extension correspondant ?
Rajouter les *plug-ins* `org.eclipse.jdt.core` et `org.eclipse.core.resources` aux dépendances du *plug-in* (dans l'onglet *Dependencies* du descriptif), puis utiliser l'option *Update the classpath and the compiler compliance settings* dans l'onglet *Overview* pour mettre à jour le chemin d'accès aux classes.

Le morceau de code suivant génére un objet de type `SourceCodeInfoList` d'après les fichiers sources Java présents dans les projets ouverts de l'espace de travail :

```
1  public SourceCodeInfo createSCI(String name, File file) {
2      int numberOfLines = 0, numberOfCommentLines = 0;
3      if (file.getAbsolutePath().endsWith(".java")) {
4          try {
5              BufferedReader reader = new BufferedReader(new FileReader(file));
6              String line;
7              while ((line = reader.readLine()) != null) {
8                  line = line.trim();
9                  numberOfLines++;
10                 if (line.startsWith("//") || line.startsWith("/*")
11                  || line.startsWith("*/") || line.startsWith("*"))
12                     numberOfCommentLines++;
13             }
14         } catch (FileNotFoundException e) {
15         } catch (IOException e) {
16         }
17     }
18     return new SourceCodeInfo(name, numberOfLines, numberOfCommentLines);
19 }
20
21 public void updateSCIList(IJavaElement element,
22                           SourceCodeInfoList list) {
23     try {
24         if (element instanceof IJavaProject) {
25             IPackageFragmentRoot[] roots = ((IJavaProject) element).getPackageFragmentRoots();
26             for (int i = 0; i < roots.length; i++)
27                 if (!roots[i].isArchive())
28                     updateSCIList(roots[i], list);
29         } else if (element instanceof IPackageFragmentRoot
30                 || element instanceof IPackageFragment) {
31             IJavaElement[] children = ((IParent) element).getChildren();
32             for (int i = 0; i < children.length; i++)
33                 updateSCIList(children[i], list);
34         } else if (element instanceof ICompilationUnit) {
35             ICompilationUnit unit = (ICompilationUnit) element;
36             list.add(createSCI(unit.getPath().toPortableString(),
37                                unit.getResource().getLocation().toFile()
38             ));
39         }
40     } catch (JavaModelException e) {
41     }
42 }
43
44 public SourceCodeInfoList createSCIList() {
45     SourceCodeInfoList list = new SourceCodeInfoList();
46     IProject[] projects = ResourcesPlugin.getWorkspace().getRoot().getProjects();
47     for (int i = 0; i < projects.length; i++)
48         if (projects[i].isOpen()).ssi
```