Exercice Rose/Java : génération de code et rétro-conception Java

Développer avec Rational Rose, v3.0-fr
Octobre 2001

1. Pré-requis

Cet exercice simple a été testé avec Rose 2001A (Java Add-In 5.0) et le JDK 1.3.1 de Sun sous Miscrosoft Windows 2000 SP1.

Note : un JDK 1.2.x fera aussi l'affaire ; remplacer dans ce cas les « 1.3.1 » ci-dessous par le numéro de la version du JDK installé.

- Le JDK 1.3.1 doit être installé (par exemple dans `D:\jdk1.3.1`).
- La variable d'environnement Windows `%JAVA_HOME%` doit être définie, et avoir le chemin ci-dessus comme valeur, typiquement :
  ```
  %JAVA_HOME%=D:\jdk1.3.1
  ```
- `%JAVA_HOME%\bin` doit être dans le PATH.
- `%JAVA_HOME%\src.jar` doit être expansé sur place. Pour cela, depuis un prompt DOS :
  ```
  D:
  cd D:\jdk1.3.1
  jar -xf src.jar
  ```
- Cela crée un répertoire `D:\jdk1.3.1\src` avec les fichiers sources des bibliothèques du JDK.

2. Conventions

Les explications générales, les objectifs sont indiqués en gras.

Les actions à effectuer sont indiquées en "normal".

Les actions menu, boîte de dialogue sont indiquées soulignées.

`Les noms de fichiers, de variables "pathmap" et d'éléments UML sont indiqués en caractères "télétype".`

3. Etapes à suivre

3.1. Préparation

- Vérifier les pré-requis et lancer Rose en ne sélectionnant aucun *framework* si certains sont proposés.
- Sélectionner avec Add-Ins : Add-In Manager l'Add-In Java 5.0 (au passage, vous pouvez désactiver d'autres Add-Ins, cela allégera le menu Tools).
- Dans Tools : Options..., tab Notation, sélectionner `Java` comme `Default Language`.
- Créer si ce n'est déjà fait un répertoire de travail pour l'exercice.

- Avec File : Edit Path Map..., créer une variable Rose `$JDK_SRC` pour le répertoire `%JAVA_HOME%\src`. Créer également une variable Rose `$JAVA_EXO_FR` pour le répertoire de travail.
- Sauver le modèle (vide pour le moment) dans `$JAVA_EXO_FR/zoo.mdl`.

3.2. Création d'un modèle sur le thème du ZOO...

- Créer 2 paquetages dans la vue logique : `animal` et `zoo`, avec la dépendance suivante :

```
zoo ---> animal
```

- Dans le paquetage `animal`, créer une classe abstraite `Animal` spécialisée en `Girafe` et `Ours`, deux classes concrètes.
- Note : pour rendre la classe `Animal` abstraite, vous avez deux moyens. Le fait d'avoir Java sélectionné comme langage par défaut a changé la boîte de dialogue qui apparaît après un double-clic sur une classe ; vous pouvez donc utiliser cette nouvelle boîte de dialogue, ou bien utiliser la boîte de dialogue classique à laquelle on accède par Clic-droit + Open Standard Specification ....
- Dans `Animal`, créer un attribut privé `nom : String` et une opération publique `dormir()`.

3.3. Première génération de code...

- Dans le diagramme de classes `Class Diagram : Logical View / Main`, sélectionner le paquetage `animal` et générer les sources Java avec Tools : Java/J2EE : Generate Code.
- Une boîte de dialogue Assign CLASSPATH Entries apparaît, dont la colonne de gauche n'indique pas a priori le répertoire voulu pour la génération de code. Effectivement, on est allé un peu vite : il faut préciser *où* le code doit être généré.
- Cliquer sur Edit....
- Une boîte de dialogue Project Specification apparaît. Il faut modifier le chemin de recherche de manière à avoir `$JAVA_EXO_FR` en premier dans le chemin de recherche (que Rose/J utilise pour l'affectation de package UML à des répertoires de packages Java). Note : cette boîte de dialogue s'obtient également avec Tools : Java/J2EE : Project Specification....
- Vérifier avec la sélection/déselection de "Resolve Pathmap Variables" comment se résout `$JAVA_EXO_FR`.
- Ajouter aussi `$JDK_SRC` juste en dessous.
- Cliquer OK pour fermer la boîte de dialogue Project Specification.
- Continuer maintenant la génération du code pour le paquetage `animal`, en sélectionnant à gauche `$JAVA_EXO_FR` et à droite `animal`.
- Il y a des *warnings*... Regarder la fenêtre de *log* qui signale quelque chose comme :

```
WARNING: Class Logical View::animal::Animal - no
return type has been specified for operation dormir
- default will be used
```

  Ne rien modifier dans le modèle pour l'instant.
- Visiter le diagramme de classes pour le paquetage `animal`.
- Visiter le source généré pour `Animal` en faisant : Clic-droit + Java/J2EE : Edit Code....
- Voici quelques notes à propos de ce source :
  - Ont bien été générés :

        - l'attribut privé `nom` de type `String`,
        - l'opération publique `dormir()`, avec un corps vide.
    - `dormir()` retourne `void`, alors qu'on ne l'avait pas précisé dans le modèle : voir le *warning* ci-dessus.
    - En bonus le constructeur par défaut `Animal()`.
- Fermer l'éditeur Java.
- Constater que le diagramme par défaut du package `animal` reflète maintenant la synchronisation entre le code et le modèle : le constructeur par défaut a été ajouté à chacune des classes.

3.4. Premières règles de génération de code...

Il y a toujours plusieurs stratégies pour générer du code à partir d'un modèle ; ces stratégies se choisissent en paramétrant des propriétés de génération de code.

- S'assurer que rien n'est sélectionné dans les diagrammes.
- Regarder les propriétés par défaut pour la génération de code avec Tools : Options et en sélectionnant l'onglet Java.
- Examiner le groupe de propriétés « Project » (champs « Type » dans la boîte de dialogue) :
    - *ClassPath* reprend les entrées de Tools : Java/J2EE : Project Specification...
    - *DefaultOperationReturnType* vaut *void*, ce qui explique le `void` généré par défaut pour l'opération `dormir()`.
    - *Editor* vaut *BuiltIn* ce qui explique que l'éditeur Java de Rose (et non celui d'un IDE ou autre) est appelé (voir dans la documentation comment configurer Rose pour avoir l'éditeur de votre choix).
- Examiner le groupe de propriétés « Class » :
    - *GenerateDefaultConstructor* vaut *True*, c'est pour cela que l'on avait bénéficié de la génération gratuite du constructeur `Animal()` ; changer pour *False*.
- Cliquer sur OK. Note : certaines de ces options sont également accessibles depuis Tools : Java/J2EE : Project Specification... dans l'onglet Code Generation.
- Répercuter la même modification sur les propriétés de génération de code liées localement à la classe `Animal`.

3.5. Etoffons un peu le modèle...

- Les girafes et les ours ont leur manière bien à eux de dormir. En UML, cela se traduit par :
    - `dormir()` est une opération abstraite pour `Animal`.
    - `Ours` et `Girafe` doivent réimplémenter `dormir()`.
- Pour rendre `Animal.dormir()` abstraite :
    - Clic-droit + Open Standard Specification... sur `Animal`.
    - Clic sur l'onglet Operations.
    - Double-clic sur `dormir`.
    - Clic sur l'onglet Java.
    - Changer la propriété *Abstract* en *True*.

  - OK.
  - OK.
- Une autre façon plus simple de faire, spécifique à l'Add-In Java :
  - Sélectionner la classe Animal dans le diagramme et faire Clic-droit + Select In Browser
  - Expanser la classe `Animal` dans le *Browser* jusqu'à voir l'opération `Animal.dormir()`.
  - Double-cliquer ensuite dessus.
  - Examiner la boîte de dialogue qui apparaît.
- Remarque : pour des raisons historiques, la version actuelle de Rose/J oblige à entrer cette information une deuxième fois pour voir apparaître l'opération abstraite en italique dans le diagramme (cette info « opération abstraite » est décorélée entre le modèle et les propriétés de génération de code Java) :
  - Sélectionner la classe `Animal` dans le diagramme de classes
  - Clic-droit + Open Standard Specification...
  - Clic sur l'onglet Details
  - Cocher la case Abstract.

Cela sera corrigé dans une prochaine version de Rose/J.

- Pour redéfinir `dormir()` dans `Ours` et `Girafe`, il faut simplement ajouter une nouvelle opération publique `dormir()` dans chaque classe... Astuce : utiliser le copier/coller, mais attention au caractère abstrait de l'opération. Ou encore, utiliser Clic-droit + New Operation.
- Modifier le constructeur par défaut de `Animal` en créant `Animal(nom : String)` ; le rendre *protected*.
- Faire de même avec un constructeur public `Ours(nom : String)` dans `Ours` et l'équivalent dans `Girafe`.
- Ajouter un accesseur public `getNom() : String` à `Animal`.
- On peut vérifier que tout est OK en sélectionnant les trois classes dans le diagramme (Ctrl-A) et en générant le code puis en examinant les fichiers sources depuis Rose.

3.6. Un peu de code Java...

En faisant Browse Code... depuis Rose, compléter le code généré avec les indications suivantes :

- `Animal.java` :

```
protected Animal(String nom) {
    this.nom = nom;
}
public String getNom() {
    return this.nom;
}
```

- `Girafe.java` :

```
public Girafe(String nom) {
    super(nom); // appel au constructeur de Animal
}
public void dormir() {
```

```
        System.out.println(  "Je suis la girafe "
                           + this.nom
                           + " et je dors debout...");
    }
```

- `Ours.java` :

  Comme `Girafe.java`, mais un ours dort en boule au lieu de dormir debout...

3.7. Discussion sur le statut de la classe `String` (et première rétro-conception de code)

La classe `String` est une classe Java prédéfinie dans le paquetage `java.lang`. En rétro-ingéniérie, par défaut, elle apparaît donc comme une classe à part entière. Essayons :

- Sélectionner la classe `Animal`.
- Clic-droit + Java : Reverse Engineer Java...
- Sélectionner le fichier source de `Animal` ou faire Select All.
- Reverse puis attendre...

Normalement rien ne se passe !

En effet, par défaut, Rose/J considère `String` comme un type primitif Java, ce qu'elle n'est pas en réalité. Ce paramétrage est modifiable via :

- Tools : Java/J2EE : Project Specification...
- Onglet Fundamental Types.

Si `String` n'était pas présente dans cette liste (faire l'essai si vous le souhaitez en modifiant la liste), la rétro-conception aurait transformé l'attribut nom en association unaire vers la classe `String`. (Rappelez-vous la proximité sémantique en UML entre un attribut et une association...)

3.8. Un zoo héberge *plusieurs* animaux...

En UML, pour exprimer ce *plusieurs* :

- Visiter le paquetage `zoo` (qui pour l'instant est resté vide).
- Y créer la classe `Zoo`.
- Par *drag & drop*, importer la classe `Animal` vers le diagramme.
- Créer une association unidirectionnelle de `Zoo` vers `Animal`, avec côté `Animal` une multiplicité `0..* :`
  - Clic-droit + Multiplicity : Zero or More
- et un rôle (public) `pensionnaire` :
  - Clic-droit + Role name et entrer « +pensionnaire ».

Dans chaque langage de programmation, il y a plusieurs stratégies pour exprimer la multiplicité `0..*` (qui signifie *est un ensemble de* ou *est une liste de)*. La stratégie adoptée par Rose est exprimée à travers les propriétés de génération de code.

- Sélectionner la classe `Zoo`.
- Clic-droit + Java/J2EE : Generate Code en mettant le package `zoo` dans `$JAVA_EXO_FR`.

Vérifier que le code généré inclut :

```
public class Zoo {
    public Animal pensionnaire[];
}
```

C'est la propriété de génération par défaut, mais nous allons la modifier :

- Double-clic sur l'association.
- Sélectionner l'onglet Java A.
- La valeur de la propriété *ContainerClass* est vide, y taper *java.util.Vector*.
- Mettre *new java.util.Vector()* comme *InitialValue*.
- OK.
- Régénérer le code pour `Zoo`.
- Examiner le code généré.

Le comportement par défaut du générateur de code (lorsque la valeur de *Container* est vide) a été modifié.

3.9. Un petit programme ?

Disons que le zoo est surveillé par un gardien qui, le soir, demande aux animaux de dormir.

- Créer la classe `Gardien` dans le paquetage `zoo`.
- Créer une association unidirectionnelle de `Gardien` vers `Zoo`, avec un nom de rôle `+surveillant` et une multiplicité égale à 1 du côté de `Gardien`. (Un gardien surveille un zoo.)
- Dans un nouveau diagramme `Dependencies` de `zoo`, créer les dépendances suivantes :

  ```
  Gardien ----> Animal
  Gardien ----> Ours
  Gardien ----> Girafe
  ```

  (Un gardien sait parler aux animaux du zoo.)
- Créer une opération publique `direDeDormir()` pour `Gardien`.
- Créer une opération publique statique (voir propriété de génération de code dans l'onglet Java) :

  ```
  main(args : String[]) : void
  ```

  pour `Gardien` (c'est le programme principal que l'on place ici, arbitrairement).
- Générer le code Java pour `Gardien`.
- Visiter le source de `Gardien` et le compléter de la manière suivante (note : le texte source est fourni dans un fichier `Gardien.txt`) :

```
public void direDeDormir() {
  System.out.println("Allez tout le monde, on dort !");
  Animal a;
  for (java.util.Iterator i =
         this.surveillant.pensionnaire.iterator();
       i.hasNext(); ) {
        a = (Animal) i.next();
        a.dormir();
  }
}

public static void main(String[] args) {
```

```
        Ours oscar = new Ours("Oscar");
        Girafe gudule = new Girafe("Gudule");
        Girafe gertrude = new Girafe("Gertrude");
        Gardien jean_pierre = new Gardien();
        jean_pierre.surveillant = new Zoo();
        java.util.Vector tous = jean_pierre.surveillant.pensionnaire;
        tous.addElement(oscar);
        tous.addElement(gudule);
        tous.addElement(gertrude);
        jean_pierre.direDeDormir();
    }
```

- Depuis un *directory viewer* dans le répertoire `$JAVA_EXO_FR`, créer les fichiers :
  - `build.bat` (pour compiler le programme) :

```
javac -classpath . zoo\Gardien.java
pause
```

  - `run.bat` (pour exécuter le programme) :

```
java -classpath . zoo.Gardien
pause
```

Ça marche pour vous ?

3.10. Un peu plus loin (rétro-conception)

On met à notre disposition un service de réveil automatique que nous aimerions réutiliser pour réveiller les animaux du zoo (maintenant qu'ils sont endormis).

Ce service est en cours de développement sous forme d'un paquetage Java disponible dans le fichier `service-fr.zip` : mettre les `.java` et `.class` dans un sous-répertoire `service` de votre répertoire de travail (à côté des autres paquetages Java) :

- Si ce n'est pas déjà le cas, ouvrez le modèle `zoo.mdl`.
- Tools : Java/J2EE : Reverse Engineer....
- Sélectionner `service` qui se trouve dans votre répertoire de travail.
- Cliquer sur Add All.
- Cliquer sur Select All puis sur Reverse.
- Cliquer sur Done quand tout est terminé.

Le paquetage `service` existe maintenant dans le modèle. Nous allons l'utiliser :

- Mettre ce nouveau paquetage (utiliser le *drag & drop* depuis le *browser*) dans le diagramme `Main` de la vue logique.
- Visualiser le contenu du diagramme principal de ce paquetage.
- Ajouter les dépendances suivantes :

```
Reveil      ---> Dormeur
zoo         ---> service
animal      ---> service
```

- Créer un nouveau diagramme de classes `UtilisationService` dans le paquetage `animal`.
- Mettre (utiliser le *drag & drop* depuis le *browser*) les classes suivantes dans ce diagramme :

```
Animal
```

```
Dormeur
Reveil
```

- Faire que `Animal` réalise `Dormeur`.
- Puisque `Animal` est un `Dormeur` maintenant, il faut lui ajouter une opération `reveiller()`. Allez-y !

Maintenant nous avons un modèle partiel utilisant le paquetage de service.

- Sélectionner les 3 paquetages (dans le diagramme Main) et générer le code Java.
- Vérifier les changements dans le code de `Animal`.

Vérifier que le code compile toujours avec `build.bat`.

4. Conclusion

Nous espérons qu'à travers cet exercice simple, vous avez pu comprendre comment utiliser Rose/J pour la conception de programmes Java.