Interface-utilisateur graphique d'une application Java

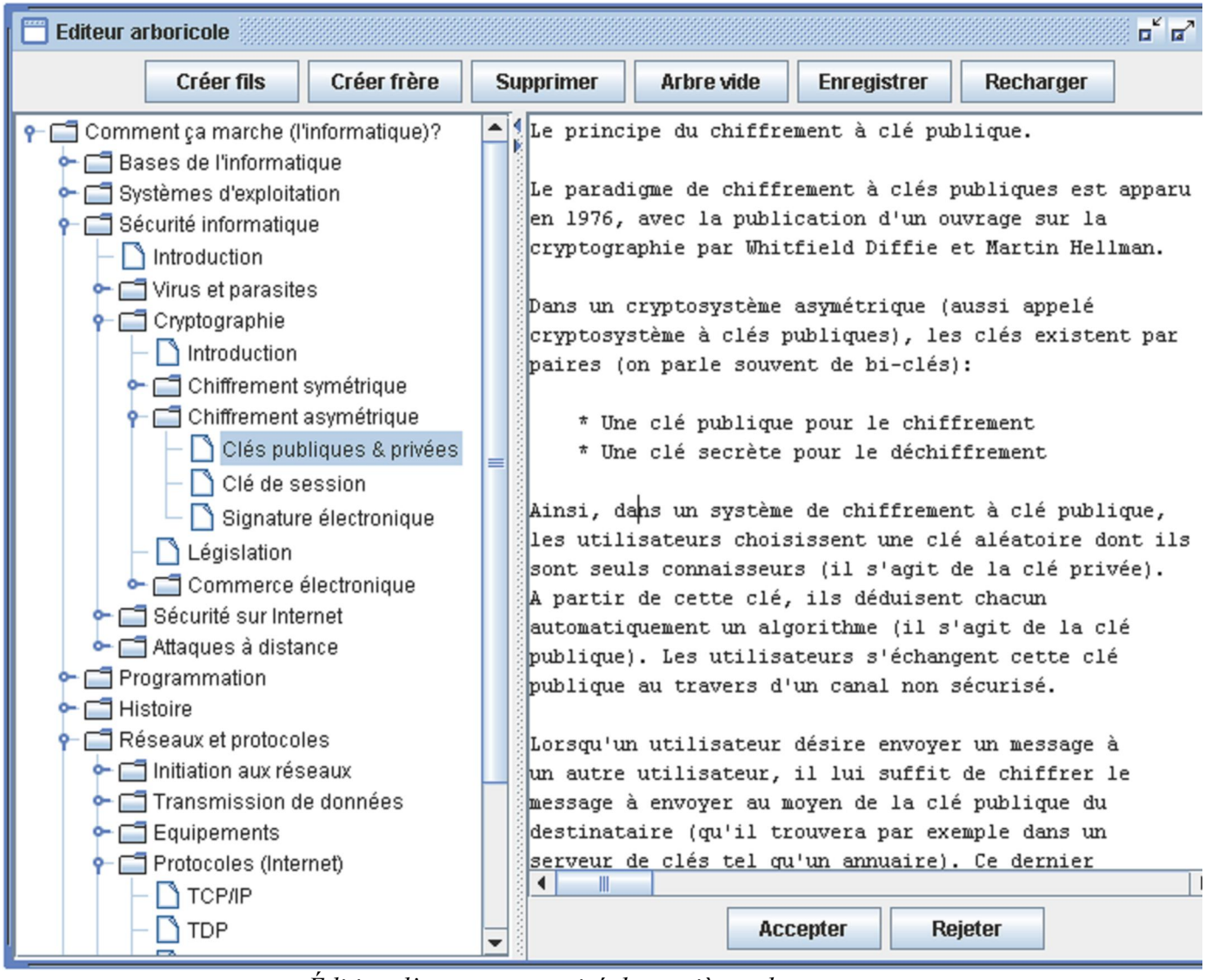
DUT Informatique 1ère année Henri Garreta

TP 5 - Éditer un texte arborescent

Objectif. L'objectif de cet exercice est la réalisation d'une application pour éditer un texte organisé sous forme d'arborescence : la racine représente le document tout entier, les fils de la racine les diverses parties dans lesquelles le document est divisé, chacune divisée à son tour en sous-parties, elles-mêmes divisées à leur tour, etc. Chaque nœud de l'arbre porte un *titre* et, facultativement, un *contenu*.

L'image ci-dessous montre l'aspect de l'application que nous voulons réaliser. Elle doit permettre d'insérer (boutons *Créer fils* et *Créer frère*) et supprimer (bouton *Supprimer*) un nœud à n'importe quel endroit de l'arbre, ainsi que de vider entièrement ce dernier (bouton *Arbre vide*), de l'enregistrer dans un fichier et de le recharger ultérieurement (boutons *Enregistrer* et *Recharger*).

Lorsqu'un nœud est sélectionné dans le volet de gauche, la zone de texte de droite montre son contenu, et il est possible alors de modifier ce dernier. Deux boutons (*Accepter* et *Rejeter*) permettent d'intégrer le texte modifié dans l'arbre ou, au contraire, de revenir à l'état du contenu avant modification.



Édition d'un texte organisé de manière arborescente

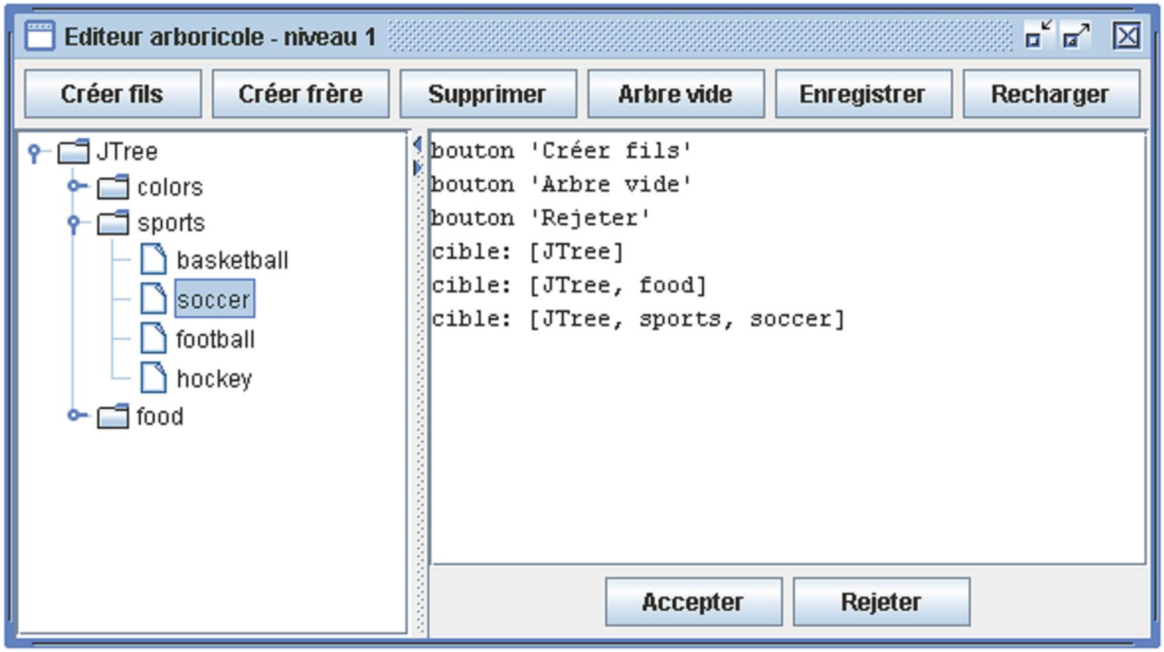
Cet exercice illustre l'utilisation de plusieurs éléments de *Swing* :

- principalement, les vues arborescentes (`JTree`) et le matériel sous-jacent (`TreeModel`, `TreeNode`, etc.),
- au passage, les panneaux partagés (`JSplitPane`), les zones de texte (`JTextArea`) et les panneaux avec des barres de défilement (`JScrollPane`),
- également la sélection d'un fichier à l'aide d'une boîte de dialogue spécialisée (`JFileChooser`),
- ainsi que l'écriture et la lecture de fichiers d'objets (ce qu'on appelle la *sérialisation* des objets).

Nous allons réaliser quatre programmes successifs, chacun représentant une étape d'un cheminement vers le programme demandé.

5.1 - Mise en place de l'interface graphique

Au premier niveau de notre application tous les composants de l'interface sont mis en place mais la structure de données – c.-à-d. l'arbre avec les morceaux de texte – n'existe pas et les boutons n'ont donc pas l'effet voulu. Voici ce qu'on obtient quand on lance l'application :



Version 1, après quelques clics...

L'arbre qui apparaît à gauche est l'arbre par défaut, « offert » par la classe **`JTree`** quand on ne lui donne pas un autre arbre à afficher. La zone de texte, inutilisée pour le moment, nous servira pour y afficher des messages qui révèlent la détection de pressions sur les boutons et de sélections dans l'arbre.

Exo 5.1.1 - EditeurArboricole, version 1

Correction

L'exercice consiste à écrire et tester la classe **`EditeurArboricole`**.

Indications. C'est une sous-classe de **`JPanel`** et elle implémente l'interface **`ActionListener`** (elle sera l'auditeur des événements produits par les boutons).

Variables d'instance de cette classe (par exemple, **`protected`**) :

- **`arbre`** (type **`JTree`**) est l'arbre affiché dans le panneau de gauche,
- **`brancheCible`** (type **`TreePath`**) représente la branche (i.e. la suite de nœuds partant de la racine) couramment sélectionnée ; par exemple (voyez la figure ci-dessus), **`[JTree, sports, soccer]`** est l'expression écrite de la branche cible lorsqu'on sélectionne le nœud étiqueté **`soccer`**,
- **`zoneTexte`** (type **`JTextArea`**) est la zone de texte du panneau de droite,
- **`boutonAccepter`**, **`boutonRejeter`** (type **`JButton`**) les deux boutons du volet de droite - on verra plus tard que, mentionnés dans plusieurs méthodes, il es nécessaire de les avoir comme variables d'instance plutôt que comme variables locales d'une méthode.

Il est conseillé de centraliser la définition des textes des boutons en en faisant une collection de constantes de classe :

```
protected static final String CREER_FILS = "Créer fils";
protected static final String CREER_FRERE = "Créer frère";
protected static final String SUPPRIMER = "Supprimer";
etc.
```

Le constructeur de cette classe fait essentiellement deux choses : initialiser l'arbre et placer les composants de l'interface graphique. L'initialisation de l'arbre ressemble à ceci :

```
...
arbre = new JTree();
arbre.setEditable(true);
arbre.setShowsRootHandles(true);
```

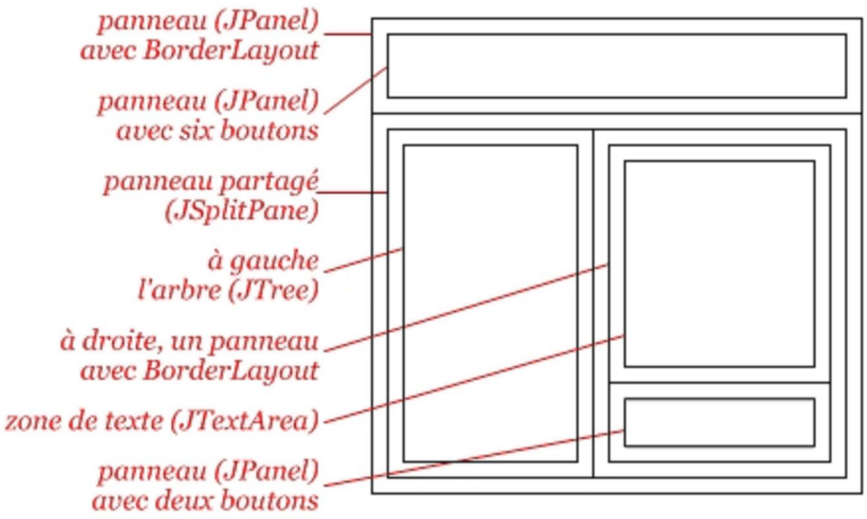
```
arbre.getSelectionModel().setSelectionMode(
    TreeSelectionModel.SINGLE_TREE_SELECTION);

arbre.addTreeSelectionListener(new TreeSelectionListener() {
    public void valueChanged(TreeSelectionEvent evt) {
        nouvelleCible(arbre.getSelectionPath());
    }
});
...
```

Explication : une fois l'arbre créé, l'appel de **`setEditable`** rend les modifications permises, l'appel de **`setShowsRootHandles`** précise que la racine a une « poignée de développement » comme les autres nœuds, enfin l'appel de **`setSelectionMode`** indique que la sélection de plusieurs nœuds n'est pas permise.

On associe ensuite à l'arbre un objet **`TreeSelectionListener`** anonyme, dans lequel la méthode **`valueChanged`** spécifie ce qu'il faut faire chaque fois que le noeud sélectionné a changé. Ici on appelle une méthode **`nouvelleCible`** (qu'il faut écrire) avec, pour argument, le chemin joignant la racine au nœud qui vient d'être sélectionné. Dans cette version du programme, la méthode **`nouvelleCible`** se limite à ajouter à la zone de texte une ligne montrant ce chemin sous forme de suite de nœuds.

La mise en place de l'interface graphique se fait comme d'habitude à l'aide de tout un micmac de panneaux et *layout managers* :



Afin de permettre à l'arbre et aux contenus textuels de grandir sans déborder on prendra soin de ne pas poser ces composants directement sur les panneaux qui les supportent, mais de les ajouter à des objets **`JScrollPane`**, eux-mêmes posés sur les panneaux en question.

Il est conseillé de rationaliser la création des huit boutons, qui sont identiques et ont le même auditeur d'événements « *action* » (à savoir, l'objet **`EditeurArboricole`** lui-même) en créant une méthode auxiliaire **`nouveauBouton`** prenant pour argument le texte du bouton.

Puisque la classe **`EditeurArboricole`** est censée implémenter l'interface **`ActionListener`** il faudra écrire aussi la méthode **`actionPerformed`** qui, pour commencer, se limitera à ajouter un texte indicatif dans la zone de texte.

Méthode principale. La méthode principale de cette classe est la méthode **`main`** « générique » souvent utilisée, qui restera la même dans les phases suivantes de cet exercice. Il s'agit de

- créer un objet cadre (**`JFrame`**),
- lui attribuer une opération de fermeture par défaut,

- lui donner un nouvel objet **`EditeurArboricole`** pour panneau de contenu,
- lui donner une taille (**`pack`**) et le rendre visible.

5.2 - Les compétences d'un *JTree*

Dans cette partie nous nous proposons d'étudier le comportement d'un arbre (classe **`JTree`**) pas, ou très peu, « personnalisé ».

En particulier, nous allons constater qu'un tel arbre, muni d'un modèle par défaut (classe **`DefaultTreeModel`**) pilotant des noeuds par défaut (classe **`DefaultMutableTreeNode`**) permet déjà de réaliser un éditeur d'arbres très convenable, du moins si on se contente de nœuds simples, portant chacun une seule information (par exemple, une étiquette).

Ainsi, à partir de l'application précédente, avec un modèle et des nœuds par défaut, il suffira d'ajouter le code qui détecte les actions sur les quatre premiers boutons pour obtenir une application permettant de faire ceci :

(Dans l'exemple précédent, la zone de texte est utilisée - comme au niveau précédent - pour afficher une trace des sélections faites.)

Exo 5.2.1 - EditeurArboricole, version 2

Correction

Voici ce qu'il faut faire à la version 1 pour obtenir la version 2 :

1. Introduire deux nouvelles variables d'instance :

- **`modele`** (de type **`DefaultTreeModel`**) pour représenter le *modèle* de l'arbre, un objet qui fait le lien entre l'arbre-structure-de-données, représenté par sa racine, et la présentation graphique qu'en fait le composant **`JTree`**,
- **`racine`** (de type **`MutableTreeNode`**) qui représente la structure de données manipulée.

N.B. L'appellation *modèle* fait référence au *design pattern* appelé « Modèle-Vue-Controleur ». La vue se charge de la présentation graphique, le contrôleur détecte et transmet les actions de l'utilisateur sur la vue, le modèle représente les données que la vue montre (ou du moins l'interface entre les données et la vue).

2. Initialiser ces variables nouvelles, par exemple au début du constructeur. Dans cet exercice nous voulons

nous en tenir aux versions par défaut des objets en jeu ; par conséquent, nous nous limiterons à remplacer

```
arbre = new JTree();
```

par

```
racine = new DefaultMutableTreeNode("Racine");
modele = new DefaultTreeModel(racine);
arbre = new JTree(modele);
```

3. Modifier **actionPerformed** pour qu'elle réagisse aux quatre premiers boutons. Par cela, des comparaisons successives de la chaîne **commande** aux constantes **CREER_FILS**, **CREER_FRERE**, **SUPPRIMER** etc. doit permettre d'appeler la méthode adéquate parmi **creerFils(String etiquette)**, **creerFrere(String etiquette)**, **supprimerNoeud()** et **supprimerTousLesNoeuds()**.

4. Enfin, il va falloir écrire les quatre méthodes mentionnées ci-dessus. A titre d'indication, voici la plus compliquée des quatre, on vous laisse chercher les autres :

```
1       private void creerFrere(String etiquette) {
2           if (brancheCible != null) {
                MutableTreeNode noeud =
3                       (MutableTreeNode) brancheCible.getLastPathComponent();
                MutableTreeNode parent = (MutableTreeNode) noeud.getParent();
4               if (parent != null) {
5                   MutableTreeNode neuf = new DefaultMutableTreeNode(etiquette);
6                   modele.insertNodeInto(neuf, parent, parent.getIndex(noeud) + 1);
7                   arbre.scrollPathToVisible(brancheCible.pathByAddingChild(neuf));
                    return;
                }
            }
8           Toolkit.getDefaultToolkit().beep();
        }
```

Quelques explications (en se référant aux numéros ci-dessus) :

1. L'argument **etiquette** (mis par la méthode appelante, c'est-à-dire **actionPerformed**) est un moyen pour créer des étiquettes provisoires, comme **"Noeud1"**, **"Noeud2"**, etc., destinées à être changées ensuite par l'utilisateur.
2. Pas question de créer un frère si aucun nœud n'est sélectionné.
3. Le nœud visé est le dernier (**getLastPathComponent**) de la branche cible. Les changements de type qui apparaissent ici et à la ligne suivante découlent de ce que les résultats des opérations **getLastPathComponent** et **getParent** sont déclarés comme des **TreeNode** (nœud d'arbre), une interface plus générale que **MutableTreeNode** (nœud *modifiable*).
4. Pas question de donner un frère à la racine (la racine est le seul nœud qui n'a pas de parent)
5. C'est ici qu'on crée un nouveau nœud, c'est donc ici qu'on choisit le type précis des nœuds (**TreeNode** et **MutableTreeNode** ne sont que des interfaces). Dans cette version, nos nœuds sont des **DefaultMutableTreeNode** (une classe concrète).
6. On insère de nœud en tant que fils de son parent. La méthode **insertNodeInto** demande le rang souhaité ; bien entendu, dans notre cas c'est le rang suivant celui du nœud visé.
7. La méthode **scrollPathToVisible** assure que tous les nœuds du chemin indiqué seront dépliés, et que le dernier sera visible dans la fenêtre. La méthode **pathByAddingChild** fabrique un chemin en ajoutant un nœud (ici **neuf**) à un chemin (ici **brancheCible**).
8. Si un des deux tests n'a pas été positif, la méthode fait entendre un son.

Écrivez les quatre méthodes **creerFils(String etiquette)**, **creerFrere(String etiquette)**, **supprimerNoeud()** et **supprimerTousLesNoeuds()** et faites-vous plaisir en constatant à quel point le programme obtenu est satisfaisant.

5.3 - Utilisation de nœuds spécifiques

Avec les nœuds par défaut fournis par la bibliothèque (**`DefaultMutableTreeNode`**) on peut faire beaucoup de choses, mais dans les applications réelles il faut souvent des nœuds spécialisés, plus étroitement dépendants de ce que l'application manipule.

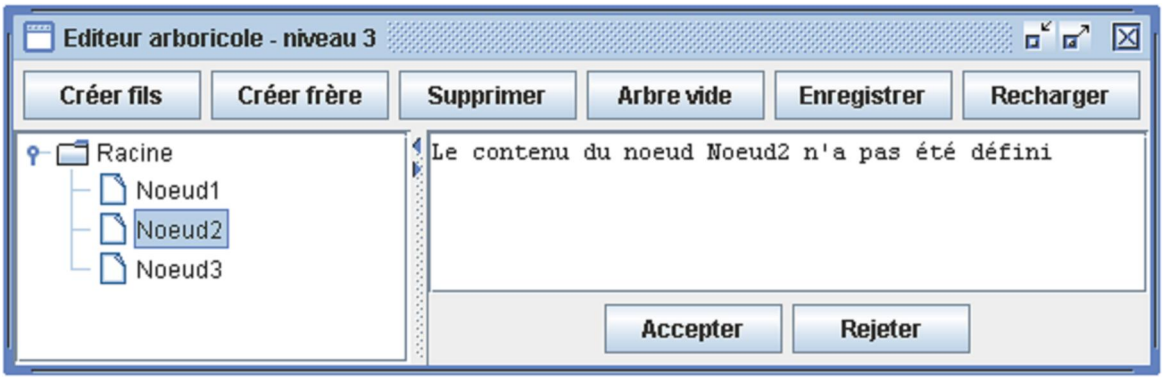
Pour illustrer cela, nous allons introduire des nœuds à peine plus complexes que les nœuds par défaut. Ce seront les instances de la classe **`NoeudTextuel`**, et chacune portera deux informations

- **`titre`** (de type **`String`**)
- **`contenu`** (de type **`String`**). Dans cette phase de l'exercice, la valeur de ce champ restera constant, égal à une valeur mise lors de l'initialisation, genre **`"Le contenu du noeud`** *titre du noeud* **`n'a pas été défini"`**

Puisqu'il s'agit de mettre en place un arbre, deux autres informations sont nécessaires dans chaque nœud :

- `listeDeFils` (de type `Vector`), un vecteur contenant les références sur les fils du nœud en question,
- `pere` (de type `NoeudTextuel`), une référence vers le nœud parent du nœud en question.

Nous ferons en sorte que la sélection d'un nœud produise l'affichage de son contenu dans la zone de texte, mais pour le moment cela ne sera pas d'un intérêt palpitant :



Exo 5.3.1 - NoeudTextuel

Correction

La classe **`NoeudTextuel`** possède

- les quatre variables d'instance mentionnées plus haut,
- le constructeur qui les initialise correctement (le **`titre`** est donné en argument, le contenu est convenu)
- deux méthodes **`getContenu`** et **`setContenu`** pour consulter et définir le contenu,
- une surcharge de la méthode **`toString`** très importante, car c'est elle qu'utilisera le **`JTree`** afin d'afficher les titres des nœuds.

La partie intéressante de la question est celle-ci : afin que le modèle puisse opérer sur des nœuds **`NoeudTextuel`** il faudra que ceux-ci soient une sorte de **`TreeNode`** et même, vu qu'on doit les modifier, de **`MutableTreeNode`**, une sous-interface de **`TreeNode`**. Il faudra donc que notre classe commence par

```
public class NoeudTextuel implements MutableTreeNode {
    ...
```

ce qui nous obligera à définir les treize méthodes imposées par **`MutableTreeNode`** (dont sept héritées de **`TreeNode`**). Vous en trouverez les spécifications dans la documentation en ligne de l'API.

La plupart de ces méthodes sont faciles à comprendre et toutes sont très courtes (la plupart font une ligne et aucune ne dépasse deux lignes). On notera que beaucoup d'opérations sur les fils d'un nœud se ramènent immédiatement à des opérations sur les vecteurs.

Deux seules remarques :

- la méthode **`setUserObject`** doit affecter la valeur du **`titre`** (cette méthode, et **`toString`** - qui joue le rôle d'opération réciproque - sont appelées par le **`JTree`** lorsqu'on édite le titre d'un nœud),
- la méthode **`insert`** doit affecter la valeur au champ **`pere`**.

Exo 5.3.2 - EditeurArboricole, version 3

Correction

Le principal changement à faire à la version 2 de la classe **`EditeurArboricole`** pour obtenir la version 3 est évident : remplacer *toutes* les occurrences de

```
new DefaultMutableTreeNode( un titre )
```

par

```
new NoeudTextuel( un titre )
```

(il y en a trois : une pour initialiser la racine, une pour créer des fils, une pour créer des frères).

Cette modification serait la seule, si on n'avait pas dit que la sélection d'un noeud doit provoquer l'affichage de son contenu. Il faudra donc aussi ajouter quelques lignes à la méthode **`nouvelleCible`**.

5.4 - Éditer les contenus, sauver l'arbre

Exo 5.4.1 - EditeurArboricole, version 4

Correction

Il ne manque plus que deux fonctionnalités à notre programme : l'édition des contenus des nœuds (boutons *Accepter* et *Rejeter*) et la sauvegarde de l'arbre dans un fichier et son rechargement ultérieur.

1. Pour commencer il faut ajouter les lignes manquantes dans la méthode **`actionPerformed`** afin que tous les boutons soient reconnus. Cela introduit trois nouvelles méthodes, qu'il va falloir écrire :

- **`accepterOuRejeterTexte(boolean accepter)`**, qui traite les actions sur les boutons *Accepter* et *Rejeter*,
- **`enregistrerArbre()`** associée au bouton *Enregistrer*
- **`rechargerArbre()`**, associée au bouton *Recharger*.

2. *Édition des contenus*. En plus de transmettre des commandes, les boutons *Accepter* et *Rejeter* joueront le rôle d'indicateurs : ces deux boutons seront *inopérants* (estompés) si et seulement si le texte montré dans la zone de texte n'a pas été modifié depuis son affichage et est donc identique à la version rangée dans l'arbre.

Lors de leur création, ces deux boutons seront donc désactivés (« activé » se dit ici *enabled*). Ensuite, il faudra détecter les modifications du texte affiché ; il suffira pour cela d'associer un auditeur **`keyListener`** au composant **`zoneTexte`**, qui se chargera d'activer les deux boutons lorsque l'utilisateur fera une frappe au clavier (attention, il faut qu'un texte soit effectivement affiché, les frappes au clavier quand la zone de texte est vide ne doivent pas avoir d'effet).

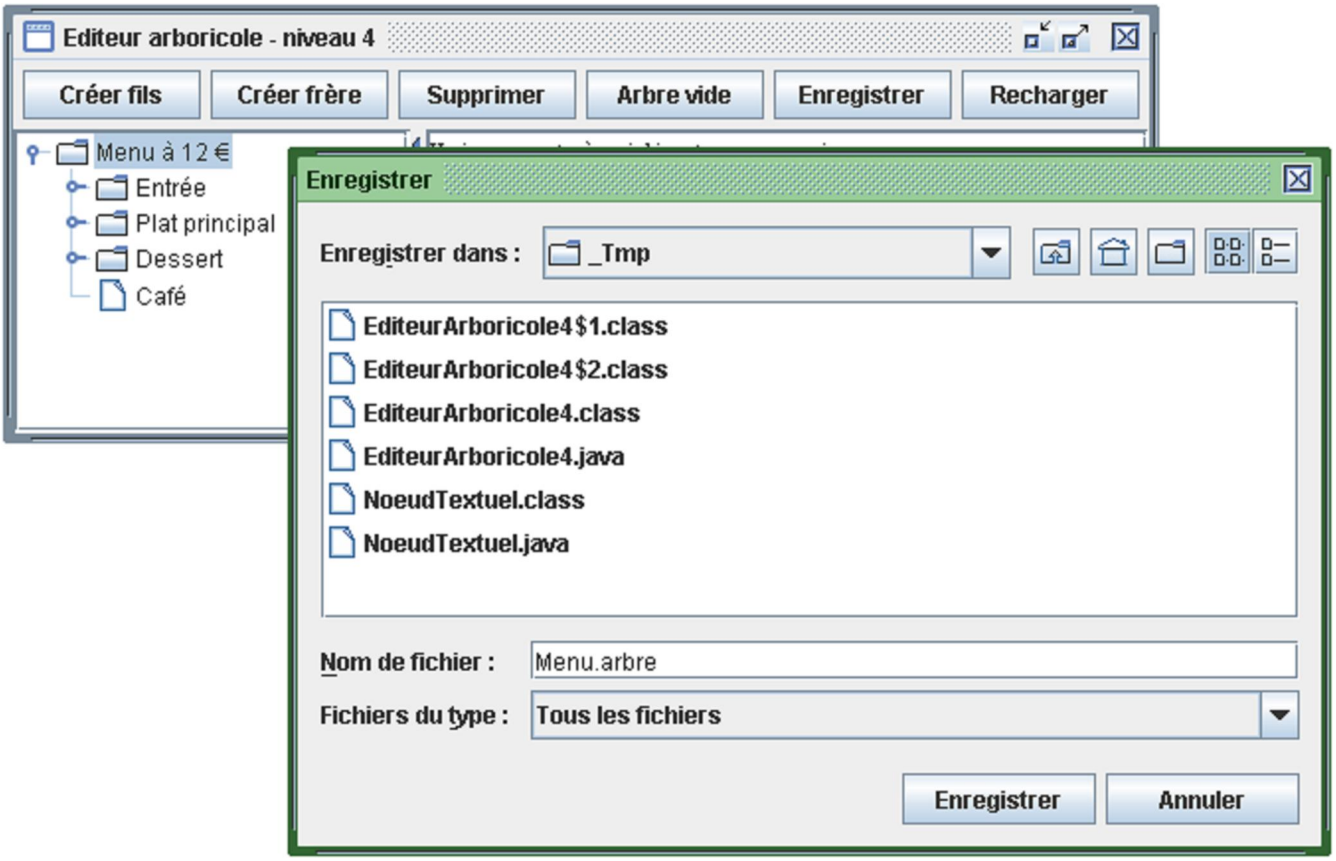
Il faudra écrire la méthode **`accepterOuRejeterTexte`** qui, avec l'argument **`true`** constitue la réponse à une pression du bouton *Accepter* et avec l'argument **`false`** à celle du bouton *Rejeter*. Cela consiste à faire jouer les méthodes **`getText`**, **`setText`** (classe **`JTextArea`**), **`getContenu`** et **`setContenu`** (classe **`NoeudTextuel`**).

Enfin, ne pas oublier d' empêcher qu'on puisse changer de sélection tant que le nœud couramment sélectionné n'est pas « sauvé » (c.-à-d. *accepté* ou *rejeté*).

3. *Enregistrement et rechargement*. Pour sauvegarder la totalité de l'arbre nous allons utiliser le mécanisme de la *sérialisation des objets*, extrêmement puissant et simple d'emploi.

Des classes et méthodes existent en Java pour enregistrer le ou les objets en question, avec assez d'information sur leurs classes pour assurer, lors des rechargements ultérieurs, la cohérence des classes utilisées lors de la sauvegarde avec celles connues lors de la restauration.

Il faut comprendre que, les objets étant toujours désignés par référence, enregistrer un objet qui a des objets membres revient à enregistrer un graphe, souvent cyclique. Cet enregistrement est loin d'être trivial !

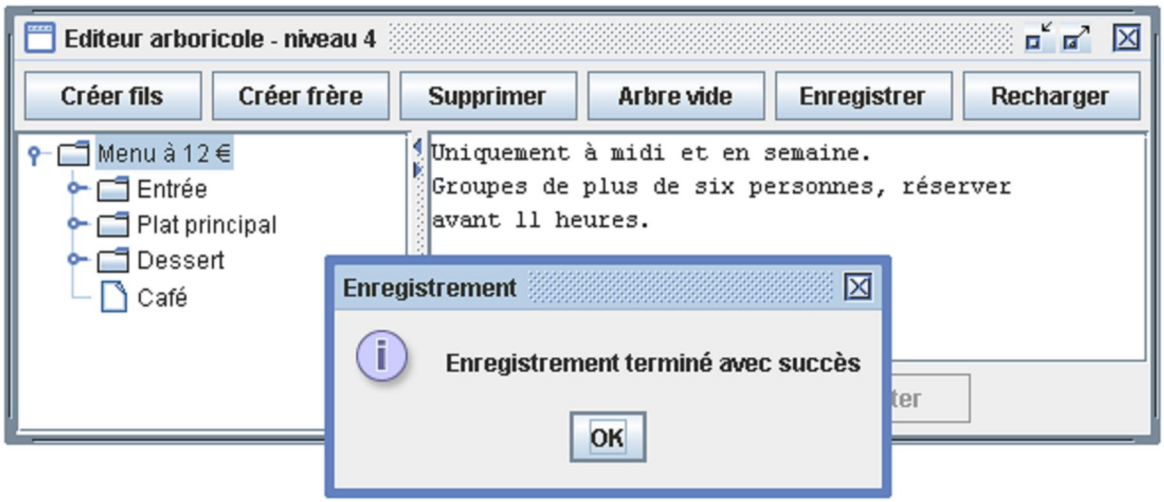


Voici la séquence d'opérations qui constituent la méthode **`enregistrerArbre`** (*mutatis mutandis*, une séquence analogue constitue la méthode **`rechargerArbre`**, on vous la laisse chercher) :

1. Creer un objet **`JFileChooser`** et appeler sa méthode **`showSaveDialog`** pour demander à l'utilisateur le nom du fichier qu'il souhaite créer.
2. Le résultat de la méthode **`showSaveDialog`** informe sur la manière dont l'utilisateur a quitté ce dialogue d'ouverture de fichier. Si ce resultat est **`APPROVE_OPTION`** c'est qu'un nom de fichier (objet **`File`**) a été spécifié avec succès.
3. A l'aide de ce nom de fichier, créer un objet **`FileOutputStream`** (c'est-à-dire un *flux de sortie aboutissant à un fichier*).
4. A l'aide de ce flux de sortie, créer un objet **`ObjectOutputStream`** (c'est-à-dire un *flux de sortie dans lequel on peut mettre des objets*).
5. Dans ce flux, écrire l'objet **`racine`** (mais oui, c'est aussi simple que cela...)
6. Fermer le flux

Un certain nombre des opérations précédentes peuvent lancer des exceptions. Il faut les attraper, et afficher une boîte de dialogue pour informer l'utilisateur (d'ailleurs, une boîte informative peut être affichée aussi

en cas de réussite) :



Attention. L'opération d'écriture d'un objet dans un **`ObjectOutputStream`** s'appelle *sérialisation* de cet objet. Pour que cette opération puisse avoir lieu il faut que l'objet en question, et tous ses objets membres, et les objets membres de leurs objets membres, etc., appartiennent à des classes déclarées implémenter l'interface **`Seriazable`**. Ici, la seule classe concernée est **`NoeudTextuel`**, dont il faudra donc écrire la première ligne sous la forme

```
public class NoeudTextuel implements MutableTreeNode, Serializable {
    ...
```

L'interface **`Serializable`** est vide, ainsi la déclaration « **`implements Serializable`** » n'engage à rien. C'est juste une marque signifiant une « permission d'être sérialisé »

La suite des TP...