Programmation Orientée Objet Avancée

S6-POO3

L3 informatique – Université de Rouen

Année 2023-2024

`http://dpt-info-sciences.univ-rouen.fr/~andarphi/lbb/index.php?category/IHM`

Philippe Andary

Table des matières

Chapitre 1 : Gestion des événements

1) Principe de gestion des événements

Toute action de l'utilisateur sur les périphériques d'entrée d'information génère des interruptions matérielles qui surgissent au niveau du noyau système ou du gestionnaire de fenêtres.

Au sein de la JVM, un processus (appelé *Toolkit Thread*) tourne en tâche de fond pour capter les signaux ainsi émis par le système et les transformer en événements, c'est-à-dire en objets contenant un ensemble d'informations en rapport avec l'origine de ces signaux. Ce processus place les événements dans une file (`EventQueue`) où ils sont stockés en attendant d'être pris en charge par un thread dédié (appelé EDT, *Event Dispatch Thread*). Les objets qui constituent le logiciel peuvent eux aussi placer des événements dans cette file d'attente.

Un seul thread (EDT, donc) est habilité à défiler les événements stockés dans cette file, ce qui impose un traitement séquentiel des événements. Lorsque EDT traite un événement, il le distribue à des objets spécifiques de l'application, dont le rôle est précisément de gérer, au moyen de méthodes adaptées, les actions à mener lorsque un événement de ce type survient. Ces objets spécifiques s'appellent des écouteurs d'événements, et ce sont eux, en définitive, qui incarnent le comportement de l'application.

Pour résumer, nous dirons que :

- Un événement logiciel est un objet qui contient toutes les informations nécessaires à la description d'un événement (au sens courant) concernant l'application.
- Un écouteur d'événements est un objet qui détient le code de gestion d'événements logiciels, réparti dans une ou plusieurs méthodes dites "méthodes réflexes" (*callback*).
- Une source d'événements logiciels est un objet susceptible d'engendrer des événements lorsqu'il est activé par l'utilisateur ou par d'autres objets au sein de l'application.

2) Catégories d'événements

On distingue trois catégories d'événements :

- Les événements dits de bas-niveau, car l'information qu'ils véhiculent est liée au système sous-jacent : périphérique d'entrée (clavier, souris, ...) ou gestionnaire de fenêtres du système.
- Les événements sémantiques, car l'information qu'ils véhiculent est liée aux fonctions abstraites du logiciels, et que ces événements sont donc porteurs de sens.
- Les événements spéciaux, qui permettent l'exécution de code spécifique sur EDT.

La première catégorie est assez restreinte, il s'agit des événements du type `java.awt.event.ComponentEvent` et de ses descendants. Pour la seconde, l'API

Java met à la disposition du programmeur un nombre impressionnant de classes d'événements. Lorsque le choix est possible, il est recommandé d'écouter les événements sémantiques plutôt que les événements de bas niveau. Enfin, la dernière catégorie est réduite aux événements de dessin (`PaintEvent`) et aux événements dits « actifs » (`ActiveEvent`) en ce sens qu'ils encapsulent du code qu'exécutera EDT.

3) Écouteurs d'événements

Puisque ce ne sont pas les objets sources eux-mêmes qui traitent l'événement, mais plutôt les écouteurs enregistrés auprès de ces sources, on parle de modèle de gestion des événements par délégation (*Delegation Event Model*).

Si `XEvent` représente le nom d'un type d'événements, les écouteurs de ce type ont un nom de la forme `XListener`. Par exemple :

- `ActionEvent` → `ActionListener`
- `DocumentEvent` → `DocumentListener`
- `MouseEvent` → `MouseListener`, `MouseMotionListener`

À chaque type d'écouteur possédant plus d'une fonction réflexe, il correspond (en général) un adaptateur qui en est une implantation triviale (les corps des méthodes sont vides) :

- `FocusListener` (2 méthodes) → `FocusAdapter`
- `WindowListener` (7 méthodes) → `WindowAdapter`

Toutefois, certains écouteurs n'ont pas d'adaptateur bien qu'ils soient dotés de plusieurs fonctions réflexes, c'est par exemple le cas de `DocumentListener` (3 méthodes, pas d'adaptateur).

L'enregistrement d'un écouteur auprès d'une source se fait par appel d'une méthode (dont le nom est de la forme `addXListener`) qui ajoute l'écouteur à la liste interne de la source mémorisant les observateurs d'événements de type `XEvent`. Plus tard, lorsque la source générera des événements de ce type, tous les écouteurs enregistrés seront notifiés.

4) Classification des événements

4.1) java.util.EventObject

`EventObject` est la classe mère de tous les types d'événements. Elle définit essentiellement une méthode `Object getSource()` qui permet d'accéder à la source de l'événement, et une méthode `String toString()` qui donne une représentation textuelle fournie de l'événement.

`EventListener` est l'interface qui lui correspond, mère de tous les types d'écouteurs. Elle ne possède aucune méthode (on parle de *marker interface*, une interface qui marque le sommet d'une hiérarchie de types, sans présumer d'un quelconque comportement pour ses sous-types).

4.2) java.awt.AWTEvent

`AWTEvent` est la classe mère des événements dits graphiques. Elle définit trois méthodes :

- **`int`** `getID()` : indique le type spécifique de l'événement ;
- `String paramString()` donne une description de l'événement utile pour déboguer ; la valeur retournée n'est jamais **`null`** ;
- **`void`** `setSource(Object)` change la source de l'événement (rarement utilisée).

La méthode `getID` est particulièrement intéressante (attention aux majuscules : `ID`, pas `Id`). Elle permet à chaque sous-type de `AWTEvent` de catégoriser les événements au sein d'un même type par la prise en compte de leur champ `id`. Ce dernier est un entier dont les valeurs, pour les événements prédéfinis de l'API, évoluent entre `COMPONENT_FIRST` (100) et `RESERVED_ID_MAX` (1999). Par exemple, pour le sous-type `MouseEvent`, le champ `id` évolue entre `MOUSE_FIRST` et `MOUSE_LAST`, ce qui permet de différencier les événements de ce type qui correspondent à l'appui sur le bouton de la souris (`MOUSE_PRESSED`), de ceux qui correspondent au relâchement du bouton (`MOUSE_RELEASED`), etc.

D'une manière générale, pour qu'un composant soit source d'événements graphiques de type `XEvent`, il faut qu'au moins un écouteur de type `XListener` soit enregistré ou alors que sa méthode `enableEvents(`**`long`**`)` ait été exécutée (à la création) avec un argument correspondant au masque des événements que l'on souhaite écouter. Les événements de type `XEvent` engendrés par le composant source auront alors un `id` compris entre `X_FIRST` et `X_LAST` et la constante associée à `id` déterminera le nom de la méthode exécutée par les `XListener`s.

`AWTEventListener` permet d'observer tous les `AWTEvent` émis par les composants AWT d'une application. Au contraire de tous les autres écouteurs, il ne doit donc pas être enregistré auprès d'un composant particulier, mais plutôt auprès d'un `Toolkit` (celui par défaut). Ce genre d'écouteur est utilisé uniquement pour déboguer une application, pour enregistrer une séquence d'événements AWT indépendamment de leur type, ou pour implanter une application accessible. Nous ne nous étendrons pas là-dessus...

4.3) java.awt.event.ComponentEvent

Avant de poursuivre, il faut faire un point sur la terminologie employée dans la documentation Java.

Un composant est "visible" (*visible* dans la documentation Java) après qu'il aura exécuté l'appel `setVisible(`**`true`**`)`. À sa construction, une fenêtre n'est pas visible ; pour tout autre composant, c'est le contraire. Après un appel à `setVisible(`**`false`**`)` un composant est invisible.

Un composant est "affichable" (*displayable* dans la documentation Java) lorsqu'il est lié à un widget du gestionnaire de fenêtres du système hôte. C'est le cas lorsque le composant est une fenêtre qui vient d'exécuter `pack()` ou `setVisible(`**`true`**`)`. C'est aussi le cas lorsque le composant n'est pas une fenêtre mais qu'il est placé dans la hiérarchie de contenance d'une fenêtre "affichable".

Un composant est "affiché à l'écran" (*showing* dans la documentation Java) s'il est affichable, visible, et si tous ses conteneurs graphiques sont visibles eux aussi.

Les événements de type `ComponentEvent` permettent de détecter les changements de visibilité, de taille ou de position d'un `Component`.

La source d'un tel événement est aussi accessible par la méthode `Component getComponent()`.

La valeur de `id` pour cet événement permet de savoir quelle méthode réflexe est appelée sur l'écouteur (qui est un `ComponentListener`) :

- `COMPONENT_MOVED` (déplacement) → `componentMoved()`
- `COMPONENT_RESIZED` (retaillage) → `componentResized()`
- `COMPONENT_SHOWN` (visibilité) → `componentShown()`
- `COMPONENT_HIDDEN` (invisibilité) → `componentHidden()`

4.4) java.awt.event.ContainerEvent

Ce type d'événements permet de détecter l'ajout ou la suppression de composants dans un `Container`.

La source d'un tel événement est aussi accessible par la méthode `Container getContainer()` et le composant fils récemment ajouté ou supprimé est accessible par `Component getChild()`.

La valeur de `id` pour cet événement permet de savoir quelle méthode réflexe est appelée sur l'écouteur (qui est un `ContainerListener`) :

- `COMPONENT_ADDED` (ajout) → `componentAdded()`
- `COMPONENT_REMOVED` (retrait) → `componentRemoved()`

4.5) java.awt.event.HierarchyEvent

Ce type d'événements permet de détecter les changements dans la hiérarchie de conteneurs parents d'un `Component`. Ce genre d'événement survient lorsque le composant, à cause de l'un de ses parents, devient affichable (*displayable*) ou non, affiché (*showing*) ou non, ou encore lorsque un conteneur de la hiérarchie du composant est déplacé ou retaillé.

La source d'un tel événement est aussi accessible par la méthode `Component getComponent()`.

Le composant qui a changé, dans la hiérarchie graphique de ce composant, est accessible par `Container getChanged()` ; le père de ce dernier est accessible par `Container getChangedParent()`.

Enfin, `long getChangedFlags()` permet de détecter, à l'aide de masques logiques, le type de modification survenue dans la hiérarchie. Les masques disponibles sont : `PARENT_CHANGED`, `SHOWING_CHANGED`, `DISPLAYABILITY_CHANGED`.

Sur des événements de ce type, on pourra utiliser deux types d'écouteurs. On peut tout

d'abord utiliser un `HierarchyListener` pour détecter les changements de structure ou d'affichabilité des éléments de la hiérarchie à laquelle appartient le composant source (`id == HIERARCHY_CHANGED` → `hierarchyChanged()`) et le masque de l'événement permet d'affiner ainsi :

- `PARENT_CHANGED` : changement de parent dans la hiérarchie du composant ;
- `DISPLAYABILITY_CHANGED` : changement de l'affichabilité de la hiérarchie du composant ;
- `SHOWING_CHANGED` : changement de l'affichabilité à l'écran de la hiérarchie du composant.

Mais on peut aussi utiliser un `HierarchyBoundsListener` pour détecter les changements de position ou de taille des conteneurs de la hiérarchie à laquelle appartient le composant source :

- (`id == ANCESTOR_MOVED` → `ancestorMoved()`) : changement de position d'un ancêtre ;
- (`id == ANCESTOR_RESIZED` → `ancestorResized()`) : changement de taille d'un ancêtre.

Remarque : quelques lignes dans la documentation Java nous expliquent qu'il n'est pas nécessaire de définir soi-même des écouteurs de type `ComponentListener`, `ContainerListener` ou `Hierarchy`[`Bounds`]`Listener` pour gérer les modifications des composants de l'API ou de leur hiérarchie, car ces mécanismes de gestion sont déjà implantés.

4.6) javax.swing.event.AncestorEvent

Ce type d'événements permet de détecter les modifications d'un conteneur parent d'un `JComponent`. Un tel événement survient lorsque un ancêtre du composant source change de visibilité (par appel à `setVisible`) ou de position à l'écran.

La source d'un tel événement est aussi accessible par la méthode `JComponent getComponent()`. De nouvelles méthodes permettent d'accéder à l'ancêtre qui a été modifié (`Container getAncestor()`) ou à son parent (`Container getAncestorParent()`).

L'écouteur `javax.swing.event.AncestorListener` est doté de trois méthodes :

- **`void`** `ancestorAdded(AncestorEvent)` : exécutée quand la source devient affichée à l'écran (par action sur elle-même ou sur l'un de ses parents) ;
- **`void`** `ancestorMoved(AncestorEvent)` : la source est déplacé (par action sur elle-même ou sur l'un de ses parents) ;
- **`void`** `ancestorRemoved(AncestorEvent)` : la source devient non affichée à l'écran (par action sur elle-même ou sur l'un de ses parents).

4.7) java.awt.event.FocusEvent

Ce type d'événements permet de détecter les changements d'état du focus, ce qui est

important ne serait-ce que parce qu'un composant se dessine différemment selon qu'il a le focus ou pas.

Mais tous les composants ne génèrent pas obligatoirement des événements de focus. Pour qu'un composant ait des chances d'avoir le focus il faut tout d'abord que `isFocusable() == true`. Par la suite, le composant peut toujours réclamer le focus avec `requestFocusInWindow()`.

Il y a deux niveaux de changement de focus :

- les changements permanents : on a donné le focus à un autre composant par appui sur la touche de tabulation, ou clic sur un autre composant ;
- les changements temporaires : on a “prêté” le focus durant un instant pour le récupérer ensuite (le temps de déplacer une barre d'ascenseur, par exemple).

Les méthodes disponibles sur ce type d'événements sont :

- `Component getOppositeComponent()` : donne l'autre composant (celui qui n'est pas la source) concerné par le changement de focus ;
- `boolean isTemporary()` : indique le niveau de changement (permanent ou temporaire) du focus.

La valeur de `id` pour l'événement permet de savoir quelle méthode réflexe est appelée sur les `FocusListener` :

- `FOCUS_GAINED` (focus obtenu) → `focusGained()`
- `FOCUS_LOST` (focus perdu) → `focusLost()`

4.8) java.awt.event.InputEvent

Il s'agit de la classe abstraite mère des événements bas niveau d'entrée de données (`KeyEvent` et `MouseEvent`).

Sa méthode `void consume()`, déclarée protégée dans `AWTEvent` et rendue publique à ce niveau, permet de marquer l'événement comme "consommé" lorsqu'on souhaite annuler son traitement. Attention toutefois, un `InputEvent` consommé par un `AWTEventListener` ne sera pas transmis aux autres écouteurs, alors qu'il le sera tout de même s'il est consommé ailleurs (!). Dans ce dernier cas, c'est aux écouteurs de prendre en compte (ou pas !) le fait que l'événement est consommé par appel à la méthode `isConsumed()`.

La méthode `long getWhen()` indique le moment où l'événement est survenu (en ms depuis le 01-01-1970 à 0 h). Les méthodes à valeur booléenne `isAltDown()`, `isAltGraphDown()`, `isControlDown()`, `isMetaDown()` et `isShiftDown()` permettent de détecter si une touche de commande a été pressée lors de la création de l'événement.

4.9) java.awt.event.KeyEvent

Permet de détecter les frappes clavier à destination du composant ayant le focus.

Plusieurs méthodes dans cette classe :

- **char** `getKeyChar()` : donne le caractère associé à l'événement ;
- **int** `getKeyCode()` : donne le code géographique de la touche associée à l'événement (`VK_A`, `VK_ENTER`, `VK_SHIFT`, ...) ;
- **int** `getKeyLocation()` : donne un code permettant de distinguer la provenance de la touche par rapport au clavier (`KEY_LOCATION_LEFT`, `KEY_LOCATION_NUMPAD`, ...).

4.9.1) Événements de type spécifique KeyEvent.KEY_PRESSED

On doit utiliser ces événements pour détecter l'enfoncement d'une touche sur le clavier. Cela permet de détecter les frappes qui n'affichent pas de caractères alphanumériques (CR, Alt, Shift, ...). Pour ces événements :

- `getKeyCode()` vaudra `VK_A`, `VK_SHIFT`, ...
- `getKeyLocation()` vaudra `KEY_LOCATION_LEFT`, `KEY_LOCATION_STANDARD`, `KEY_LOCATION_RIGHT`, ...
- `getKeyChar()` donnera le caractère frappé ou `CHAR_UNDEFINED`.

4.9.2) Événements de type spécifique KeyEvent.KEY_RELEASED

On doit utiliser ces événements pour détecter le relâchement d'une touche au clavier. Ces événements ont les mêmes caractéristiques que les précédents.

4.9.3) Événements de type spécifique KeyEvent.KEY_TYPED

On doit utiliser ces événements pour détecter les caractères entrés au clavier. Un tel événement peut correspondre à la frappe simultanée de plusieurs touches (par exemple 'A' est obtenu par la combinaison de touches Shift+a).

- `getKeyChar()` donnera le caractère entré ;
- `getKeyCode()` vaudra `VK_UNDEFINED` ;
- `getKeyLocation()` vaudra `KEY_LOCATION_UNKNOWN`.

4.10) java.awt.event.MouseEvent

Permet de détecter les événements liés à l'utilisation de la souris. On utilise un `MouseListener` pour gérer les clics et le survol de la souris sur les composants. On utilise un `MouseMotionListener` pour gérer les déplacements de la souris sur ce composant, et enfin, on utilise un `MouseWheelListener` pour gérer l'action de la molette de la souris sur le composant.

Quelques méthodes intéressantes :

- **int** `getButton()` : indique le bouton de la souris qui a été cliqué (`NOBUTTON`, `BUTTON1`, `BUTTON2`, `BUTTON3`) ;
- **int** `getClickCount()` : donne le nombre de clics effectués avec ce bouton ;
- **boolean** `isPopupTrigger()` : détecte un clic d'activation de menu surgissant (pas très portable car doit être testé dans `mousePressed()` ou dans `mouseReleased()` selon la plateforme...) ;
- `Point getLocationOnScreen()` : indique la position absolue du clic sur

l'écran ;
- `Point getPoint()` : donne la position du clic relativement à la source.

La valeur de `id` pour l'événement permet de savoir quelle méthode réflexe est appelée sur les `MouseListener`s :

- `MOUSE_CLICKED` (clic effectué) → `mouseClicked()` ;
- `MOUSE_PRESSED` (bouton appuyé) → `mousePressed()` ;
- `MOUSE_RELEASED` (bouton relâché) → `mouseReleased()` ;
- `MOUSE_ENTERED` (pointeur entre dans le composant) → `mouseEntered()` ;
- `MOUSE_EXITED` (pointeur sort du composant) → `mouseExited()`.

ou sur les `MouseMotionListener`s :

- `MOUSE_MOVED` (déplacement) → `mouseMoved()` ;
- `MOUSE_DRAGGED` (déplacement bouton enfoncé) → `mouseDragged()`.

4.11) java.awt.event.MouseWheelEvent

`MouseWheelEvent` dérive de `MouseEvent` avec de plus `getID() == MOUSE_WHEEL`. Un tel événement est généré lorsque la molette de la souris est activée et tout `MouseWheelListener` enregistré est activé via sa méthode `mouseWheelMoved()`.

5) Événements légers

Lorsque la source émet très fréquemment des notifications, ou que l'on ne souhaite pas que l'événement ne transmette d'information spécifique, on peut utiliser la classe des événements légers `javax.swing.event.ChangeEvent` qui ne donnent accès qu'à la source de l'événement.

Le cas typique d'utilisation (voir l'exemple de `JSlider`) est le suivant : la source ne crée qu'un seul événement, qu'elle mémorise et émet à chaque fois que cela est nécessaire (économie en temps et en mémoire).

La classe `ChangeEvent` nous donne la possibilité de revisiter le patron Observer. En effet, on peut utiliser l'interface `ChangeListener` au lieu de `java.util.Observer`. Le modèle n'est alors plus `Observable` et il faudra réintroduire le mécanisme manquant dans chaque modèle. Mais heureusement, la classe `EventListenerList` possède des méthodes génériques qui facilitent la tâche d'implantation de ce mécanisme.

La différence tient à ce que le modèle n'est plus passé directement en argument d'une méthode `update`, mais en tant que source d'un événement de type `ChangeEvent` communiqué à une méthode `stateChanged` : on a ainsi uniformisé le traitement des modèles avec celui des vues puisqu'on peut maintenant écouter aussi bien les premiers que les secondes.

6) Beans Java

JavaBean est une spécification Java pour des composants réutilisables et indépendants de leur environnement. Un *bean* est un composant qui, une fois compilé, peut être intégré

à chaud dans un logiciel (éditeur graphique notamment). Dit autrement, un *bean* :

- peut être inspecté dynamiquement (introspection) ;
- possède et gère des propriétés qui le caractérisent ;
- expose ses propriétés (getter, setter) ;
- notifie ses changements de valeur de propriété par événements ;
- est persistant (on peut sauvegarder et restaurer son état interne).

6.1) Propriétés des beans

Un *bean* défini des propriétés qui caractérisent son état interne ou son apparence. Une telle propriété est définie par :

- un nom ;
- un éventuel accesseur en lecture (getNom, ou isNom si la propriété est booléenne) ;
- un éventuel accesseur en écriture (setNom).

Les composants AWT et Swing sont des beans (voir par exemple la gestion de la propriété "background" de `Component`).

6.1.1) Propriétés simples

Une propriété est dite simple (*simple property*) lorsqu'elle est constituée d'une seule valeur, indépendante des autres propriétés.

Par exemple, la propriété de nom "background" dans la classe `Component` est mise en place à l'aide d'un attribut qui mémorise sa valeur (`Color background`), d'un getter permettant l'accès en lecture (`public Color getBackground()`) et d'un setter autorisant l'accès en écriture (`public void setBackground(Color)`).

6.1.2) Propriétés indexées

Une propriété est dite indexée (*indexed property*) lorsqu'elle est constituée d'une séquence de valeurs de type `X`. Si l'accès en lecture est permis, on trouvera un getter de la forme `X[] getName()` et un setter de la forme `void setName(X[] values)`. Selon le cas, il se peut aussi que l'accès soit proposé pour chaque élément (`X getName(int)` et `void setName(int, X)`).

6.1.3) Propriétés liées

Une propriété est dite liée (*bound property*) lorsque tout changement de valeur de la propriété est notifié (par événement) aux écouteurs préalablement enregistrés. Dans ce cas, l'accesseur en écriture (`setName`) génère obligatoirement des événements de type `PropertyChangeEvent` et notifie les `PropertyChangeListener` préalablement enregistrés.

6.1.4) Propriétés contraintes

Une propriété est dite contrainte (*constrained property*) lorsqu'elle est liée et que tout changement de valeur de la propriété est préalablement soumis à l'approbation d'autres

composants. Dans ce cas, l'accesseur en écriture (`setName`) doit notifier des `VetoableChangeListener` préalablement enregistrés, juste avant que la propriété ne change de valeur. Ces derniers ont la possibilité de lever une `PropertyVetoException` à partir de leur (seule) méthode réflexe `vetoableChange`. Généralement, cette exception interrompt l'exécution de `setName` et elle est transmise au code qui souhaitait modifier la propriété. Ce dernier est donc prévenu que le changement de valeur n'a pas été accepté.

6.2) java.beans.PropertyChangeEvent

Le changement de valeur de la propriété liée est décrit par :

- `Object getOldValue()` donne l'ancienne valeur de la propriété ;
- `Object getNewValue()` donne la nouvelle valeur de la propriété ;
- `String getPropertyName()` donne le nom de la propriété qui a changé.

L'interface `java.beans.PropertyChangeListener` définit une seule méthode réflexe `void propertyChange(PropertyChangeEvent)`.

Dans `Component`, on trouve les méthodes (lire `PropertyChangeListener` pour `PCL`) :

- `public void addPCL(PCL)` pour enregistrer un écouteur, de type PCL, sur les changements de toutes les propriétés ;
- `public void addPCL(String p, PCL pcl)` pour enregistrer l'écouteur `pcl` sur la propriété de nom `p` ;
- `public void removePCL(PCL pcl)` pour retirer l'écouteur `pcl` de l'écoute de toutes les propriétés ;
- `public void removePCL(String p, PCL pcl)` pour retirer l'écouteur `pcl` seulement de l'écoute de la propriété de nom `p` ;
- `public PCL[] getPCLs()` pour récupérer la liste des écouteurs de type PCL préalablement enregistrés ;
- `protected void firePropertyChange(String name, T oldValue, T newValue)` pour que le bean notifie ses écouteurs de type `PCL` à l'aide d'un événement généré pour indiquer que la propriété de nom `name` et de type `T` est passée de `oldValue` à `newValue`.

6.3) java.beans.PropertyChangeSupport

C'est une classe utilitaire qui fournit le code nécessaire pour faciliter l'implantation des propriétés liées ; on peut ainsi déléguer la gestion des `PropertyChangeEvent` à une instance de cette classe.

Elle est utilisée par `Component`, par conséquent, toutes les classes qui en dérivent peuvent réutiliser les méthodes installées dans `Component`. Pour les autres classes (nos modèles en TP, par exemple), on peut s'inspirer du code source de la classe `Component` pour réaliser soi-même la gestion des propriétés liées.

Il existe aussi une classe `javax.swing.event.SwingPropertyChangeSupport` qui permet de choisir (argument du constructeur) que les notifications se fassent sur EDT (on verra plus tard pourquoi il faut, dans certaines situations, s'assurer de cela).

6.4) java.beans.VetoableChangeListener

Ces écouteurs écoutent des `PropertyChangeEvent`. L'interface ne décrit qu'une seule méthode réflexe `void vetoableChange()`.

Dans `javax.swing.JComponent`, on trouve les méthodes (lire `VetoableChangeListener` pour `VCL`) :

- `public void addVCL(VCL vcl)` ;
- `public void removeVCL(VCL vcl)` ;
- `public VCL[] getVCLs()` ;
- `protected void fireVetoableChange(String name, T oldValue, T newValue) throws PropertyVetoException`.

6.5) java.beans.VetoableChangeSupport

C'est une classe utilitaire qui fournit le code nécessaire pour faciliter l'implantation des propriétés contraintes par rapport à la gestion des veto sur les `PropertyChangeEvent`.

Elle est utilisée par `JComponent`, par conséquent, toutes les classes qui en dérivent peuvent réutiliser les méthodes de `JComponent`. Pour les autres, on peut s'inspirer du code de `JComponent` pour réaliser soi-même la gestion des propriétés contraintes.

Attention, le rollback implémenté dans la méthode `fireVetoableChange` de cette classe est bogué jusqu'à la version 7b38 du SDK ! Je vous donne en cours une règle méthodologique d'utilisation depour tte classe...

Chapitre 2 : Gestion de la répartition des composants

1) Principes de gestion de la répartition des composants

La gestion de la répartition des composants dans un conteneur se fait à l'aide d'un gestionnaire de répartition. Il s'agit d'un objet qui demande à chaque composant du conteneur de calculer ses tailles préférée/minimale/maximale pour décider de la taille et de la position que devra prendre chaque composant au sein du conteneur. Certains composants pouvant être eux-même des conteneurs, le fonctionnement est généralement récursif sur la hiérarchie des composants graphiques de l'application.

Lorsqu'un composant graphique exécute sa méthode `invalidate` (de `Component`) il est alors marqué comme non valide (`isValid() == false`) ainsi que toute sa super-hiérarchie graphique. Cela signifie que son état a changé de telle sorte qu'il faut recalculer la taille et/ou la position de tous les conteneurs de sa super-hiérarchie graphique.

Lorsqu'un composant graphique exécute sa méthode `validate` (de `Component`) il est marqué comme valide (`isValid() == true`). Cette méthode est appelée lorsque le composant vient d'être retaillé et/ou repositionné par le gestionnaire de répartition de son conteneur parent.

Un conteneur est valide s'il est valide en tant que composant, et si tous ses sous-composants sont valides eux aussi. La méthode `validate` appliquée à un conteneur n'active son gestionnaire de répartition que si le conteneur n'est pas valide puis elle le marque comme valide, sinon elle ne fait rien.

Pour les composants Swing, on utilisera de préférence la méthode `revalidate` (de `JComponent`) qui commence par invalider le composant (et sa super-hiérarchie) puis envoie un message de validation au conteneur du sommet de la hiérarchie du composant. Cette méthode est garantie s'exécuter sur EDT, après tout événement antérieur à son appel.

Juste après sa création, un composant n'est pas encore valide. L'appel à la méthode `pack()` de `JFrame` active le gestionnaire de répartition du *content pane* de la fenêtre pour la première fois et (récursivement) celui de chaque conteneur de la hiérarchie de contenance de la fenêtre. Après exécution de cette méthode, tous les composants de la fenêtre sont donc valides. Lorsqu'on retaille la fenêtre à la souris, ou suite à l'appel de l'une des deux méthodes `validate()` (de `Component`) ou `revalidate` de (`JComponent`), la méthode `layoutContainer` de `LayoutManager` est appelée et c'est ainsi que les gestionnaires sont amenés à faire leur travail.

2) Méthodes utiles à la gestion de la répartition

2.1) Pour les gestionnaires

Pour connaître l'espace souhaitable pour un composant, le gestionnaire peut utiliser les méthodes suivantes, définies dans `Component` :

- `Dimension getMinimumSize()` ;

- `Dimension getPreferredSize()` ;
- `Dimension getMaximumSize()`.

Pour fixer la taille d'un composant il utilise `void setSize(Dimension)`, pour en fixer la position il utilise `void setLocation(Point)` et pour fixer les deux à la fois il utilise `void setBounds(int x, int y, int w, int h)` (coordonnées du coin supérieur gauche, largeur et hauteur).

2.2) Pour le programmeur

Pour connaître la taille d'un composant, le programmeur a accès aux méthodes suivantes de `Component` :

- `Dimension getSize()` ;
- `int getHeight()` ;
- `int getWidth()`.

Pour déterminer la position du composant on utilise :

- `Point getLocation()` ;
- `int getX()` ;
- `int getY()`.

Et pour configurer la taille du composant, les trois méthodes suivantes :

- `void setMinimumSize(Dimension)` ;
- `void setPreferredSize(Dimension)` ;
- `void setMaximumSize(Dimension)`.

3) Algorithmes de calcul de la taille des composants

3.1) Dimension getMinimumSize()

Voici l'algorithme de calcul utilisé par cette méthode :

```
SI minSize a été fixée[1] ALORS
  res = minSize
SINON SI c'est un composant Swing ALORS
  res = valeur calculée par le ui-delegate
SINON SI le composant est un conteneur ALORS
  res = layoutMgr.minimumLayoutSize(this)
SINON SI le composant est un composant AWT ALORS
  res = valeur calculée par le peer
SINON
  res = null
FIN SI
retourner res != null ? res : getSize()
```

1 Par appel à `setMinimumSize()`.

3.2) Dimension getMaximumSize()

Voici l'algorithme de calcul utilisé par cette méthode :

```
SI maxSize a été fixée[2] ALORS
  res = maxSize
SINON SI le composant est un composant Swing ALORS
  res = valeur calculée par le ui-delegate
SINON SI le composant est un conteneur ALORS
  res = layoutMgr2.maximumLayoutSize(this)
SINON
  res = null
FIN SI
retourner res != null ? res : new Dimension(Short.MAX_VALUE,
Short.MAX_VALUE)
```

3.3) Dimension getPreferredSize()

Voici l'algorithme de calcul utilisé par cette méthode :

```
SI prefSize a été fixée[3] ALORS
  res = prefSize
SINON SI le composant est un composant Swing ALORS
  res = valeur calculée par le ui-delegate
SINON SI le composant est un conteneur ALORS
  res = layoutMgr.preferredLayoutSize(this)
SINON SI le composant est un composant AWT ALORS
  res = valeur calculée par le peer
SINON
  res = null
FIN SI
retourner res != null ? res : getMinimumSize()
```

3.4) Exemples de valeurs pour quelques composants Swing :

`JComponent`

taille minimum : `getSize()` (initialement (0, 0) !)
taille préférée : taille minimum
taille maximum : `(Short.MAX_VALUE, Short.MAX_VALUE)`

`AbstractButton` et `JLabel`

taille minimum : taille préférée
taille préférée : taille de l'icone + taille du texte + marges
taille maximum : taille préférée

`JTextField`

taille minimum : marges
taille préférée : taille du texte + marges

2 Par appel à `setMaximumSize()`.
3 Par appel à `setPreferredSize()`.

taille maximum : `(Integer.MAX_VALUE, Integer.MAX_VALUE)`

4) Installation d'un gestionnaire

Pour un `JPanel`, le gestionnaire par défaut (défini lorsqu'on ne donne pas d'argument au constructeur) est un `FlowLayout` (centré). Mais d'une manière générale, on passe le gestionnaire de répartition souhaité comme argument au constructeur du conteneur (comme dans `new JPanel(new GridLayout(2, 3))` par exemple).

Pour associer un gestionnaire à un conteneur après que ce dernier aura été construit, on peut utiliser la méthode `setLayout(LayoutManager)` de `Container`. La séquence d'instructions à utiliser est la suivante :

```
JPanel p = new JPanel(null);
p.setLayout(new XxxLayout(...));
```

5) L'interface java.awt.LayoutManager

Cette première interface spécifie les gestionnaires à fonctionnement simple. Les gestionnaires qui en dérivent directement travaillent avec les méthodes suivantes :

- `void addLayoutComponent(String, Component)` : initialisation de la gestion du composant lorsqu'il est repéré par un nom (cette méthode est appelée par les méthodes `add` de la classe `Container`, à deux arguments, dont l'un est une `String`) ;
- `void layoutContainer(Container)` : effectue la répartition des composants sur le conteneur donné en argument (cette méthode est appelée par les méthodes `validate`, `revalidate` ou lors d'un redimensionnement de la fenêtre racine à l'aide de la souris) ;
- `Dimension minimumLayoutSize(Container)` : calcule la taille minimale du conteneur donné en argument (cette méthode est appelée par le gestionnaire lui-même) ;
- `Dimension preferredLayoutSize(Container)` : calcule la taille préférée du conteneur donné en argument (cette méthode est appelée par le gestionnaire lui-même) ;
- `void removeLayoutComponent(Component)` : retire le composant du gestionnaire (cette méthode est appelée par les méthodes `remove` de `Container`).

On remarquera qu'il est nécessaire de passer le conteneur géré par le gestionnaire en argument des méthodes qui réalisent les calculs de répartition car le gestionnaire ne mémorise pas le conteneur auquel il est associé.

5.1) java.awt.FlowLayout

(*Voir cours MPOO2*)

Ce type de gestionnaires définit des constantes d'alignement absolues (`LEFT`, `CENTER`, `RIGHT`) ou relatives à l'orientation choisie (`LEADING`, `TRAILING`). Lorsque l'orientation

des composants est gauche-droite : `LEADING` ↔ `LEFT` et `TRAILING` ↔ `RIGHT`, sinon c'est le contraire : `LEADING` ↔ `RIGHT` et `TRAILING` ↔ `LEFT`.

Il existe aussi une méthode d'alignement des composants sur leur ligne de base qui est explicitée en cours sur quelques exemples.

5.2) java.awt.GridLayout

(*Voir cours MPOO2*)

L'algorithme de transformation des paramètres du constructeur (`rows` et `cols`) est le suivant :

```
Si rows > 0 Alors
    cols = (nComponents + rows - 1) / rows
Sinon
    rows = (nComponents + cols - 1) / cols
Fin Si
```

Le remplissage se fait de gauche à droite puis de haut en bas. On peut indiquer 0 ligne ou 0 colonne mais pas les deux en même temps. S l'on indique 0 ligne, le nombre de colonnes ne varie pas et le nombre de lignes est ajusté. Si l'on indique 0 colonne, le nombre de lignes ne varie pas et le nombre de colonnes est ajusté.

6) L'interface java.awt.LayoutManager2

Cette seconde interface (qui dérive la précédente) spécifie des gestionnaires plus complexes, qui utilisent des contraintes pour configurer le placement des composants. En plus des méthodes de l'interface `LayoutManager`, ils utilisent les méthodes suivantes :

- **`void`** `addLayoutComponent(Component, Object)` : initialisation de la gestion du composant (cette méthode est appelée par les méthodes `add` à deux arguments de `Container`) ;
- **`float`** `getLayoutAlignmentX(Container)` : valeur d'alignement par rapport à l'axe des X (cette méthode est appelée par le gestionnaire lui-même) ;
- **`float`** `getLayoutAlignmentY(Container)` : valeur d'alignement par rapport à l'axe des Y (cette méthode est appelée par le gestionnaire lui-même) ;
- **`void`** `invalidateLayout(Container)` : invalide le gestionnaire, permettant le recalcul des tailles minimale, préférée et maximale (cette méthode est appelée par les méthodes `invalidate`, `revalidate` ou lors du retaillage de la fenêtre à la souris) ;
- `Dimension maximumLayoutSize(Container)` : calcule la taille maximale du conteneur (cette méthode est appelée par le gestionnaire lui-même).

6.1) java.awt.BorderLayout

(*Voir cours MPOO2*)

Ce type de gestionnaire définit des contraintes de placement absolues : `NORTH`, `SOUTH`, `WEST`, `EAST` ou relatives à l'orientation horizontale du conteneur : `PAGE_START`,

`PAGE_END`, `LINE_START`, `LINE_END`.

Pour une orientation gauche-droite (`getComponentOrientation() == LEFT_TO_RIGHT`) `LINE_START` ↔ `WEST` et `LINE_END` ↔ `EAST`, sinon `LINE_START` ↔ `EAST` et `LINE_END` ↔ `WEST`. La composante verticale de l'orientation n'est pas prise en compte actuellement...

6.2) java.awt.CardLayout

Ce type de gestionnaires affiche un seul composant à la fois, mais on peut choisir celui qui est affiché :

- `void first(Container)` affiche le premier composant du conteneur ;
- `void next(Container)` affiche le composant suivant du conteneur (cyclique) ;
- `void previous(Container)` affiche le composant précédant du conteneur (cyclique) ;
- `void last(Container)` affiche le dernier composant du conteneur ;
- `void show(Container, String)` affiche sur le conteneur le composant dont le nom est passé comme second argument.

Son utilisation peut devenir fastidieuse s'il faut rajouter à la main un mécanisme de sélection du composant à afficher. `JTabbedPane` est plus simple d'utilisation et fournit le même genre de services. Un exemple est donné en cours...

6.3) java.awt.GridBagLayout

Avec ce style de gestionnaire, de loin le plus complexe et le plus complet, le conteneur est découpé selon une grille sur laquelle chaque composant se place dans une zone d'affichage qui peut couvrir plusieurs cellules.

À chaque composant est associé une contrainte qui définit précisément les rapports entre le composant et sa zone d'affichage et entre cette dernière et la grille.

6.3.1) Mise en place

À la création, seul un constructeur sans argument est disponible. On commence par associer le conteneur à son gestionnaire :

```
JPanel p = new JPanel(new GridBagLayout());
```

Puis on ajoute chaque composant, avec sa contrainte, `p.add(comp, const)` où `const` est de type `GridBagConstraints`. Le gestionnaire stocke alors cette association dans une `Hashtable<Component, GridBagConstraints>`.

6.3.2) java.awt.GridBagConstraints

La classe `GridBagConstraints` définit les contraintes appliquées par le gestionnaire au composant. Il s'agit d'un enregistrement dont les champs ont la signification suivante :

- `gridx`, `gridy` : position de la zone sur la grille ;
- `gridwidth`, `gridheight` : taille de la zone ;
- `insets` : marges du composant à l'intérieur de sa zone ;

- `anchor` : disposition du composant dans sa zone ;
- `ipadx`, `ipady` : marges internes du composant ;
- `fill`, `weightx`, `weighty` : distribution de l'espace supplémentaire.

6.3.3) Positionnement de la zone d'affichage

Cette propriété est configurée en donnant des valeurs aux attributs `gridx` et `gridy` de l'instance de `GridBagConstraints` associée au composant à disposer :

- **`int`** `gridx` (valeur ≥ 0, vaut `RELATIVE` par défaut) : numéro de colonne (débute en 0) de la cellule qui contient le coin supérieur gauche de la zone d'affichage (`RELATIVE` signifie à la suite du précédent).
- **`int`** `gridy` (valeur ≥ 0, vaut `RELATIVE` par défaut) : numéro de ligne (débute en 0) de la cellule qui contient le coin supérieur gauche de la zone d'affichage (`RELATIVE` signifie à la suite du précédent).

6.3.4) Taille de la zone d'affichage

Cette propriété est configurée en donnant des valeurs aux attributs `gridwidth` et `gridheight` de l'instance de `GridBagConstraints` associée au composant à disposer :

- **`int`** `gridwidth` (valeur ≥ 0, vaut 1 par défaut) : nombre de cellules occupées par la zone horizontalement.
- **`int`** `gridheight` (valeur ≥ 0, vaut 1 par défaut) : nombre de cellules occupées par la zone verticalement (`RELATIVE` signifie jusqu'à l'avant dernière cellule, `REMAINDER` signifie jusqu'à la dernière cellule).

6.3.5) Marges entre le composant et sa zone d'affichage

Cette propriété est configurée en donnant des valeurs à l'attribut `insets` de l'instance de `GridBagConstraints` associée au composant à disposer :

- `Insets insets` (vaut **`new`** `Insets(0, 0, 0, 0)` par défaut) : espace à rajouter à l'intérieur de la zone, autour du composant, dans l'ordre : nord, ouest, sud, est.

6.3.6) Marges internes au composant

Cette propriété est configurée en donnant des valeurs aux attributs `ipadx` et `ipady` de l'instance de `GridBagConstraints` associée au composant à disposer :

- **`int`** `ipadx` (vaut 0 par défaut) : espace à rajouter à la taille minimum du composant, répartie horizontalement de chaque côté (en parts égales).
- **`int`** `ipady` (vaut 0 par défaut) : espace à rajouter à la taille minimum du composant, répartie verticalement de chaque côté (en parts égales).

6.3.7) Ancre du composant dans sa zone d'affichage

Cette propriété est configurée en donnant des valeurs à l'attribut `anchor` de l'instance de `GridBagConstraints` associée au composant à disposer :

- **`int`** `anchor` (vaut `CENTER` par défaut) : ne sert que lorsque la zone est plus grande que le composant, indique alors l'emplacement du composant dans sa zone.

En position absolue : `NORTHWEST`, `NORTH`, `NORTHEAST`, `WEST`, `CENTER`, `EAST`, `SOUTHWEST`, `SOUTH` ou `SOUTHEAST`. Et relativement à l'orientation : `FIRST_LINE_START`, `PAGE_START`, `FIRST_LINE_END`, `LINE_START`, `CENTER`, `LINE_END`, `LAST_LINE_START`, ... Et enfin, par rapport à la ligne de base : `ABOVE_BASELINE_LEADING`, `ABOVE_BASELINE`, `ABOVE_BASELINE_TRAILING`, `BASELINE_LEADING`, `BASELINE`, `BASELINE_TRAILING`, ...

6.3.8) Étirement du composant sur sa zone d'affichage

Cette propriété est configurée en donnant des valeurs à l'attribut `fill` de l'instance de `GridBagConstraints` associée au composant à disposer :

- **`int`** `fill` (vaut `NONE` par défaut) : ne sert que lorsque la zone est plus grande que le composant, indique alors le mode d'étirement du composant dans sa zone.

Peut prendre les valeurs `NONE` (pas d'étirement), `HORIZONTAL` (étirement horizontal seulement), `VERTICAL` (étirement vertical seulement) ou `BOTH` (étirement dans les deux sens).

6.3.9) Attribution de l'espace supplémentaire disponible dans le conteneur

Cette propriété est configurée en donnant des valeurs aux attributs `weightx` et `weighty` de l'instance de `GridBagConstraints` associée au composant à disposer :

- **`double`** `weightx` (valeur ≥ 0, vaut 0 par défaut) : indique comment répartir l'espace horizontal supplémentaire du conteneur entre les différentes zones.
- **`double`** `weighty` (valeur ≥ 0, vaut 0 par défaut) : indique comment répartir l'espace vertical supplémentaire du conteneur entre les différentes zones.

Le principe du calcul des poids est le suivant. Tout d'abord le gestionnaire calcule les poids des lignes et des colonnes ainsi :

> poids d'une ligne = max des `weighty` des composants de la ligne ;
> poids d'une colonne = max des `weightx` des composants de la colonne.

Ensuite il calcule la taille préférée du conteneur. Si cette taille est plus petite que la taille courante du conteneur (par exemple suite à un retaillage) l'espace en trop est réparti entre les colonnes et les lignes proportionnellement à leur poids. Mais comment répartir les poids lorsqu'une zone recouvre plusieurs cellules ?

Je vous propose une méthode rigoureuse et infaillible pour la définition des poids. Il faut attribuer des poids entiers, les plus petits possibles à chaque ligne et chaque colonne. Ensuite, on choisit une "zone caractéristique" (unique) pour chaque ligne de la grille et une

zone caractéristique pour chaque colonne de la grille. Puis on affecte le poids en Y de chaque ligne au composant de sa zone caractéristique (définition des `weighty`) et le poids en X de chaque colonne au composant de sa zone caractéristique (définition des `weightx`). Il ne reste qu'à mettre à zéro tous les `weightx` et les `weighty` des autres composants (ceux qui ne sont pas placés sur des zones caractéristiques).

Cas particulier : lorsqu'on ne peut pas définir de zone caractéristique pour une ligne ou une colonne. La solution consiste à ajouter une zone caractéristique fantôme que l'on associe à un `JComponent`. Ce dernier bien qu'invisible puisque de taille (0, 0), nous permet néanmoins de définir cette zone caractéristique.

Récapitulatif[4]

1. Dessiner le conteneur sur une feuille de papier.
2. Dessiner une zone par composant, chaque zone s'étirant au-delà du composant afin de couvrir globalement un rectangle.
3. Quadriller le dessin en fonction des zones, numéroter les lignes et les colonnes en partant de 0 (ce qui permettra de définir `gridx` et `gridy`), puis définir les poids pour chaque ligne et chaque colonne (ce qui permettra de définir `gridwidth` et `gridheight`).
4. Pour chaque zone, se demander si son composant doit être étiré horizontalement ou verticalement (`fill`). Dans la négative, comment doit-il être aligné (`anchor`) ?
5. Déterminer les marges intra (`ipadx`, `ipady`) et extra (`insets`) composant si nécessaire.
6. Appliquer les règles pratiques de calcul des poids (`weightx`, `weighty`) en ajoutant éventuellement des fantômes.
7. Coder les contraintes à l'aide d'une classe mieux adaptée que `GridBagConstraints` (voir ci-dessous), et les rassembler dans une structure de données adaptée (par exemple un tableau) permettant d'alléger l'écriture du code.
8. Stocker les contraintes dans un tableau, une `Map`, ... toute collection permettant un traitement ultérieur simplifié (*i.e.* à l'aide d'une boucle) des contraintes.

Voici une classe `GBC` plus simple d'utilisation que `GridBagConstraints` dont elle dérive :

```
public class GBC extends GridBagConstraints {
    public GBC() {
        super();
    }
    public GBC(int gridx, int gridy) {
        this.gridx = gridx;
        this.gridy = gridy;
    }
    public GBC(int gridx, int gridy, int gridw, int gridh) {
        this(gridx, gridy);
```

4 Adapté de Horstman & Cornell, Core Java 2, volume 1 – Fundamentals, 7th Edition, 2004, Prentice-Hall.

```
        this.gridwidth = gridw;
        this.gridheight = gridh;
    }
    public GBC anchor(int a) {
        this.anchor = a;
        return this;
    }
    public GBC fill(int f) {
        this.fill = f;
        return this;
    }
    public GBC insets(int v) {
        return insets(v, v, v, v);
    }
    public GBC insets(int top, int left, int bottom, int right) {
        this.insets = new Insets(top, left, bottom, right);
        return this;
    }
    public GBC ipad(int ipadx, int ipady) {
        this.ipadx = ipadx;
        this.ipady = ipady;
        return this;
    }
    public GBC weight(double weightx, double weighty) {
        this.weightx = weightx;
        this.weighty = weighty;
        return this;
    }
}
```

Et voici un exemple de méthode `placeComponents` :

```
private void placeComponents() {
    GBC[] constraints = new GBC[] {
            // un
            new GBC().fill(GBC.BOTH),
            // deux
            new GBC(1, 0, 2, 1).fill(GBC.BOTH),
            // trois
            new GBC(0, 1, 2, 1).fill(GBC.BOTH).weight(0, 1),
            // quatre
            new GBC(2, 1, 1, 2).fill(GBC.BOTH).weight(2, 0),
            // cinq
            new GBC(0, 2).fill(GBC.BOTH),
            // six
            new GBC(1, 2).fill(GBC.BOTH).weight(1, 2)
    };
    JPanel p = new JPanel(new GridBagLayout()); {
        for (int i = 0; i < BT_NB; i++) {
            p.add(buttons[i], constraints[i]);
```

```
            }
        }
        frame.add(p, BorderLayout.CENTER);
    }
```

6.4) javax.swing.BoxLayout

Ce type de gestionnaires permet de disposer les composants selon un axe horizontal ou vertical en respectant des contraintes d'alignement ainsi que la taille maximale des composants. Les contraintes sont stockées dans les composants eux-mêmes (méthodes de `Component`) :

- **`void`** `setAlignmentX(`**`float`**`)` indique la valeur d'alignement vertical du composant : elle représente la proportion de surface du composant visible à gauche de l'axe du conteneur.
- **`void`** `setAlignmentY(`**`float`**`)` indique la valeur d'alignement horizontal du composant : elle représente la proportion de surface du composant visible en haut de l'axe du conteneur.
- **`void`** `setMaximumSize(Dimension)`.

Dans un conteneur doté d'un `BoxLayout`, chaque composant est étiré (ou pas) en fonction de la manière dont sa largeur maximale est calculée.

6.4.1) Montées et descentes

La valeur d'alignement d'un composant est donnée par :

- **`float`** `getAlignmentX()` ;
- **`float`** `getAlignmentY()`.

Elle est comprise entre 0.0f et 1.0f, et elle vaut 0.5f par défaut pour un `JComponent`. On peut la définir à l'aide des méthodes :

- **`void`** `setAlignmentX(`**`float`**`)` ;
- **`void`** `setAlignmentY(`**`float`**`)`.

Une montée est définie comme le produit d'une des dimensions par la valeur d'alignement correspondante. Une descente comme la différence entre une dimension et sa montée. Par exemple :

```
Dimension d = getPreferredSize();
int ascent = (int) (d.width * getAlignmentX());
int descent = d.width - ascent;
```

6.4.2) Axe du conteneur

L'axe d'un conteneur est une ligne fictive sur laquelle sont alignés les composants par rapport à leur propre valeur d'alignement. La position de l'axe vertical d'un conteneur doté d'un `BoxLayout` vertical est donnée en pourcentage de sa largeur par la formule : $max(ascent_i) / [max(ascent_i) + max(descent_i)]$. C'est le `BoxLayout` qui calcule lui-même cette valeur lors de l'appel à la méthode `getLayoutAlignmentX(p)`.

6.4.3) Mise en place

À la création, seul le constructeur `BoxLayout(Container, int)` à deux arguments est disponible. Le second argument indique le type d'axe au moyen de constantes absolues : `X_AXIS` ou `Y_AXIS`, ou relatives (à la propriété `componentOrientation`) : `LINE_AXIS` ou `PAGE_AXIS`.

On commence par l'association entre le conteneur et son gestionnaire :

```
JPanel p = new JPanel(null);
p.setLayout(new BoxLayout(p, X_AXIS));
```

Puis on ajoute chaque composant sans contrainte avec la méthode `add(Component)` de `Container`.

6.4.4) javax.swing.Box

On préférera l'usage de `Box` qui est un conteneur transparent prédoté d'un `BoxLayout`. De plus, cette classe permet de créer des glues, composants invisibles de dimensions minimale et préférée (0, 0) et de dimension maximale :

- avec `Box.createGlue()` : `(Short.MAX_VALUE, Short.MAX_VALUE)`
- avec `Box.createHorizontalGlue()` : `(Short.MAX_VALUE, 0)`
- avec `Box.createVerticalGlue()` : `(0, Short.MAX_VALUE)`

Elle permet aussi de créer des *struts*, composants invisibles de dimensions :

- avec `Box.createHorizontalStrut(int width)` :
  - min et pref : `(width, 0)`
  - max : `(width, Short.MAX_VALUE)`
- avec `Box.createVerticalStrut(int height)` :
  - min et pref : `(0, height)`
  - max : `(Short.MAX_VALUE, height)`

Elle permet enfin de créer des aires rigides, composants invisibles de dimensions fixées :

- avec `Box.createRigidArea(Dimension d)` :
  - min, pref, max : `d`

ou encore des composants invisibles totalement configurables :

- avec `new Box.Filler(Dimension, Dimension, Dimension)`

6.5) javax.swing.OverlayLayout

Ce type de gestionnaires est une généralisation à trois dimensions du type précédent. Il permet d'empiler des composants les uns au-dessus des autres en respectant des contraintes d'alignement ainsi que la taille maximale des composants. Ici encore, les contraintes sont stockées à l'intérieur des composants :

- `void setAlignmentX(float)` ;
- `void setAlignmentY(float)` ;
- `void setMaximumSize(Dimension)`.

6.5.1) Point axial

Le point axial de chaque composant a pour coordonnées les montées courantes du composant dans chaque direction. Les points axiaux des composants sont alignés les uns avec les autres. La position du point axial du conteneur est fonction des montées et des descentes des composants dans chaque direction.

6.5.2) Mise en place

À la création, seul un constructeur prenant le conteneur en argument est disponible. On commence par associer le conteneur avec son gestionnaire

```
JPanel p = new JPanel(null);
p.setLayout(new OverlayLayout(p));
```

Puis on ajoute chaque composant sans contrainte avec la méthode `add(Component)` de `Container`.

6.6) javax.swing.GroupLayout

La présentation que j'en fait suit celle donnée dans le tutoriel de Sun au bout du lien : `http://download.oracle.com/javase/tutorial/uiswing/layout/group.html`.

Ce type de gestionnaires permet de définir indépendamment les dispositions horizontale et verticale des composants dans le conteneur. Il sert essentiellement à créer des formulaires.

Chaque composant est enregistré deux fois dans le gestionnaire : une fois pour sa disposition horizontale, une autre pour sa disposition verticale.

6.6.1) Structurer la répartition

On définit un groupe comme un ensemble de composants, de groupes ou d'écarts (*gaps*).

Un groupe permet de répartir ses éléments séquentiellement : les uns à la suite des autres, selon l'une des deux directions horizontale ou verticale, ou parallèlement : empilés les uns au-dessus des autres, selon l'une des deux directions horizontale ou verticale.

6.6.2) Mise en place

À la création, seul un constructeur prenant le conteneur en argument est disponible. On commence par associer le conteneur et son gestionnaire :

```
JPanel p = new JPanel(null);
GroupLayout lmp = new GroupLayout(p);
p.setLayout(lmp);
```

Puis on ajoute chaque composant deux fois... sur le gestionnaire, pas sur le conteneur ! On utilise pour cela les méthodes suivantes de `GroupLayout` :

- `setHorizontalGroup(GroupLayout.Group)` ;
- `setVerticalGroup(GroupLayout.Group)` ;
- `Group.addComponent(Component)`.

Voir en cours la manière particulière d'utiliser ces méthodes.

6.6.3) Écarts

Un écart est un composant invisible, de taille fixée, que l'on intercale entre deux composants voisins, ou entre un composant et le bord de son conteneur. On peut définir ses propres écarts avec les méthodes :

- `addGap` de `GroupLayout.Group` ;
- `addPreferredGap` de `GroupLayout.SequentialGroup` ;
- `addContainerGap` de `GroupLayout.SequentialGroup`.

Mais en général, on utilise des écarts prédéfinis dans `GroupLayout` :

- `setAutoCreateGaps(boolean auto)` ;
- `setAutoCreateContainerGaps(boolean auto)`.

6.6.4) Retaillage des composants

Par défaut, le gestionnaire respecte les tailles (min, pref, max) des composants, mais on peut modifier ce comportement en ajoutant des contraintes de taille lors de l'utilisation de la méthode `addComponent(Component c, int min, int pref, int max)` de `GroupLayout.Group`. Deux constantes particulières sont disponibles : `GroupLayout.DEFAULT_SIZE` impose l'utilisation de la taille correspondante (min, pref ou max) du composant et `GroupLayout.PREFERRED_SIZE` impose l'utilisation de la taille préférée du composant à la place de la taille correspondante.

6.6.5) Lier la taille de plusieurs composants

À l'intérieur d'un même groupe parallèle, on peut lier la taille des composants en reportant à l'infini la taille maximale des éléments du groupe. Le groupe lui-même peut être retaillable ou non *via* la méthode `createParallelGroup(int align, boolean resizable)` de `GroupLayout`.

Entre deux groupes indépendants on utilisera plutôt les méthodes `linkSize([int axis,] Component... c)` de `GroupLayout`.

6.6.6) Modification de la composition du conteneur à l'exécution

La méthode `replace(Component oldComp, Component newComp)` de `GroupLayout` permet de remplacer `oldComp` par `newComp`.

La méthode `setHonorsVisibility(boolean)` de `GroupLayout` indique si, globalement, l'espace occupé par des composants devenus invisibles doit être pris en compte ou non. Il est possible de raffiner ce comportement au niveau de chaque composant avec une méthode de même nom, à deux arguments dont le premier est un composant.

6.7) javax.swing.SpringLayout

Ce type de gestionnaires permet de définir la position et la taille des composants à l'aide de ressorts (*springs*) disposés entre les arêtes des composants.

Un ressort définit une contrainte entre deux arêtes d'un même composant, ou de deux composants, ou d'un composant et de son conteneur...

7) Gestionnaires personnalisés

7.1) Pas de gestionnaire

On peut aussi ne pas utiliser de gestionnaire et placer les composants de manière absolue. Il faut donc empêcher l'association d'un gestionnaire avec le conteneur :

```
JPanel p = new JPanel(null);
```

ou encore `p.setLayout(null);` si la création du conteneur est déjà passée.

Il faut aussi donner des positions et des tailles aux composants en utilisant les méthodes de `Component` :

- `setSize` ;
- `setLocation` ;
- `setBounds`.

Il faut enfin appeler `repaint` sur chaque composant après avoir calculé sa taille et sa position.

Si le conteneur est retaillé c'est un vrai casse-tête ! On aura donc intérêt à empêcher le retaillage de la fenêtre (propriété `resizable` à **`false`**).

7.2) Autre exemple

Voir la page suivante :

https://docs.oracle.com/javase/tutorial/uiswing/layout/custom.html

Chapitre 3 : Threads

1) Généralités sur la gestion des processus

Un processus est une séquence d'instructions exécutées par une machine (un processeur), en utilisant un espace mémoire dédié et d'autres ressources fournies par le système d'exploitation. Une machine physique peut gérer plusieurs processus :

- de manière simultanée lorsque la machine possède plusieurs processeurs,
- ou de manière concurrente lorsqu'elle partage son temps de travail entre les processus,

chaque processus étant géré indépendamment des autres. Notez bien qu'une machine peut travailler à la fois de manière simultanée et concurrente.

Un processus est défini au niveau du système par :

- un identificateur (*pid*) ;
- un jeu de registres qui lui permettent de manipuler le compteur ordinal, le pointeur de pile, etc. ;
- une zone de mémoire virtuelle privée regroupant :
  - la séquence d'instructions à exécuter (zone de code) ;
  - les variables et constantes globales (zone de données) ;
  - les variables locales et les adresses de retour d'appel (pile d'exécution) ;
  - la zone d'allocation dynamique (tas) ;
- l'ensemble des ressources système qui lui sont allouées (descripteurs de fichiers, ports, etc.) ;
- un certain nombre d'informations complémentaires (priorité, etc.).

Qu'il y ait plusieurs processeurs ou non, le système réalise l'exécution concurrente des processus en partageant son temps CPU à l'aide d'un module du noyau que l'on appelle l'ordonnanceur (*scheduler*, en anglais). Ce module est nécessaire car le rapport processus/processeur est généralement supérieur à 1 ; c'est lui qui, par l'utilisation d'algorithmes sophistiqués assurera une utilisation optimale des processeurs ressources.

Par le passé, les ordonnanceurs fonctionnaient sur le mode coopératif (Windows de 3.1 à Me, Mac OS 5, ...), mode dans lequel l'ordonnanceur espère que le processus qui a la main finira par la rendre quand il aura épuisé son quota de temps. Ce mécanisme a prouvé son inefficacité (blocage infini du système d'exploitation en cas de blocage du processus élu) et les systèmes modernes (Windows de NT à aujourd'hui, Unix et ses dérivés, Mac OS X, ...) sont dorénavant dotés d'ordonnanceurs préemptifs au sein desquels le processus est évincé à la fin d'un temps prédéfini.

Le système s'interrompt donc à intervalles réguliers pour déterminer si le processeur doit changer de processus à exécuter. Dans l'affirmative :

- le système exécute l'algorithme d'ordonnancement ;
- il sauvegarde toutes les données associées au processus évincé ;
- il restaure toutes les données associées au processus élu.

On notera que ce mécanisme est coûteux en temps (par exemple, le coût direct du

changement de contexte a été mesuré à environ 1,5 µs pour un Haswell Core i7-4771[5]).

Deux processus communiquent entre eux par envoi de signaux mais aussi par l'intermédiaire d'autres mécanismes fournis par le système :

- tubes de communication ;
- fichiers ;
- communications inter processus (IPC) ;
- etc.

Ces mécanismes sont complexes et leur description sort du cadre de ce cours (voir le cours de système sur ce sujet).

Comme un processus, un thread est une séquence d'instructions exécutées par la machine, mais un thread s'exécute à l'intérieur d'un processus. Un processus peut ainsi contenir plusieurs threads. Tout thread peut accéder à la zone de mémoire et aux ressources de son processus hôte, c'est la différence fondamentale entre thread et processus : le fait que la mémoire sur laquelle ils travaillent est partagée par d'autres (cas des threads) ou pas (cas des processus).

La communication entre deux threads (frères) se fait par l'intermédiaire des variables qu'ils partagent (zone de données et tas du processus hôte).

Chaque thread exécute sa propre séquence d'instructions et dispose :

- d'un accès exclusif au jeu de registres du processus hôte lorsqu'il est en cours d'exécution ;
- d'une pile d'exécution personnelle.

Lorsque l'ordonnanceur évince un thread pour en élire un autre :

- le système exécute l'algorithme d'ordonnancement ;
- il sauvegarde les valeurs des registres et la pile d'exécution du thread qui perd la main (évincé) ;
- il restaure les valeurs des registres et la pile d'exécution du thread qui prend la main (élu).

On observe donc que le changement de contexte entre threads, étant plus simple, est plus rapide qu'entre processus.

2) Contrôleurs de threads Java

En général, plusieurs threads s'exécutent dans la JVM :

- « main thread » : thread qui exécute la méthode main de l'application.
- « event dispatch thread » : thread qui assure la gestion événementielle de Java ainsi que l'affichage des composants graphiques.
- « toolkit thread » : thread qui traduit les événements système en événements Java.
- « user threads » : threads créés, dans le code source, par le programmeur.

5 source : https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

Un thread est la conceptualisation d'une séquence d'instructions en cours d'exécution, il ne peut donc pas être directement représenté par un objet ; toutefois, on peut lui associer un objet Java qui permettra de le contrôler (démarrer, endormir, ...). En Java, un contrôleur de thread est donc une instance de la classe `java.lang.Thread`, par abus de langage on dira souvent “thread” en pensant “contrôleur de thread”. On prendra donc bien garde de ne pas mélanger ces deux notions.

Pour créer un contrôleur de thread, en Java, il faut :

1. définir le code à exécuter par le thread dans une méthode `run` :
    1. qui peut être celle d’un objet `Runnable`, ou bien
    2. qui peut être celle d’un objet `Thread`
2. instancier le contrôleur de thread pour le code précédemment défini.

Une question se pose donc : quand hériter de `Thread` et quand implémenter `Runnable` ?

- On héritera de `Thread` lorsqu'on souhaite raffiner le comportement des contrôleurs de threads (très rare).
- On implémentera `Runnable` quand on souhaite uniquement décrire le code cible d'un thread (très fréquent).

Les deux méthodes `run` (celle de la classe `Thread` et celle de l'interface `Runnable`) sont reliées entre elles du fait que l’on trouve, dans la classe `Thread`, la définition suivante :

```
public void run() {
    // target est un attribut de type Runnable
    // initialisé avec l'argument passé au constructeur
    if (target != null) {
        target.run();
    }
}
```

Le cycle de vie d'un thread démarre lors de la création de son contrôleur, il est alors dans l'état `Thread.State.NEW` (valeur retournée par la méthode `getState()`) et la méthode `isAlive()` retourne **`false`** tant que le thread n'a pas été démarré. Une fois démarré, elle retourne **`true`** et le thread oscille entre les quatre états suivants :

- `RUNNABLE` : il est élu ou évincé par le système, mais pas bloqué ni en attente ;
- `TIMED_WAITING` : il est en attente sur appel de `wait`, `join` ou `sleep` ;
- `WAITING` : il est en attente infinie sur appel de `wait` ou `join` ;
- `BLOCKED` : il est attente infinie de l'obtention d'un verrou ;

puis, à la fin de l'exécution de sa méthode `run()`, le thread est dans l'état `TERMINATED` et `isAlive()` retourne de nouveau **`false`**.

Attention, la méthode `getState()` retourne une valeur qui ne peut pas être utilisée dans la situation où un thread veut connaître l'état d'un autre thread. Sa principale raison d'être tient à ce qu'elle peut permettre de tracer, déboguer... elle correspond à l'état interne du

contrôleur de thread par rapport à la machine Java mais ne peut pas être mise en correspondance avec les états du thread tels que gérés par le système d'exploitation.

2.1) Démarrer un thread

On démarre un thread en appelant sa méthode `void start()`. Le premier appel à cette méthode lance la méthode `run()` du thread de manière asynchrone : ce dernier prend vie et `isAlive()` retourne `true` durant toute l'exécution de la méthode `run()`. Si `start()` est appelée plus d'une fois sur le thread, une `IllegalMonitorStateException` est levée.

2.2) Interrompre un thread

La méthode `void interrupt()` positionne le drapeau d'interruption d'un contrôleur à `true`. Cela signifie que l'on envoie « une demande d'attention au plus tôt de la part du thread ».

La méthode `boolean isInterrupted()` retourne la valeur du drapeau d'interruption du thread, alors que la méthode statique `boolean interrupted()` retourne la valeur du drapeau d'interruption du contrôleur du thread courant et positionne immédiatement ce drapeau à `false`.

Lorsque vous interrompez un thread, il ne vous reste plus qu'à attendre qu'il s'en rende compte : par des appels à `isInterrupted` ou `interrupted` judicieusement placés dans votre code, ou lorsque votre thread est bloqué par une méthode bloquante par la capture de l'exception levée.

Une méthode bloquante peut stopper momentanément l'exécution du thread courant dans l'attente de la réalisation d'une certaine condition, puis reprendre l'exécution normale du thread. Exemples d'opérations bloquantes :

- Les opérations d'entrées/sorties bloquantes.
- Dans la classe `Thread`, les méthodes `sleep` et `join`.
- Dans la classe `Object` les méthodes `wait`.

Si l'on interrompt un thread alors qu'il était bloqué par l'une des méthodes `wait`, `sleep` ou `join`, la méthode détecte l'interruption, durant le blocage, le plus rapidement possible. Ensuite, elle remet à `false` le drapeau d'interruption du contrôleur de thread puis elle lève une `InterruptedException`. Lors d'une opération d'entrée/sortie bloquante sur un flux classique (`java.io`) on obtient en fait une `InterruptedIOException` (à noter que cela ne fonctionne pas sous Windows : il n'y a pas d'interruption dans ces cas).

2.3) Endormir le thread courant

La méthode statique `void sleep(long)` endort le thread courant. Par conséquent, un thread ne peut pas endormir un autre thread, tout ce qu'il peut faire, c'est s'endormir lui-même... On peut détecter le thread en cours d'exécution (ou thread courant) à l'aide de la méthode statique `Thread currentThread()`.

2.4) Proposer de passer la main

La méthode statique `void yield()` laisse une chance aux autres threads de prendre la main en plaçant `currentThread()` dans le pool de threads éligibles à l'exécution. Notez que cette méthode est nécessaire si l'implantation des threads est coopérative...

2.5) Attendre la fin d'un autre thread

Les méthodes d'instance `void join([long millis])` placent le thread courant en attente de la mort du thread associé au contrôleur cible de l'appel. Ces méthodes sont bloquantes et interruptibles, et l'attente peut-être bornée par le nombre de millisecondes souhaité.

2.6) Terminer un thread

Un thread se termine naturellement quand la méthode `run()` de son contrôleur se termine. On dit alors que le thread est mort, et l'état du contrôleur est `Thread.State.TERMINATED`. Dans cet état `isAlive()` retourne `false`.

2.6.1) Méthodes stop

Attention, il n'est pas possible de terminer un thread à la fois de manière sûre et immédiate : il faut laisser au contrôleur de thread le temps d'organiser la mort du thread.

Les méthodes `stop()` de la classe `Thread` terminent immédiatement un thread, mais elles sont déconseillées (*deprecated*). En effet, ces méthodes arrêtent immédiatement (arrêt préemptif) l'exécution du code associé au thread. Par conséquent, le contrôleur de thread n'a pas le temps de “faire le ménage” avant la mort du thread et il peut laisser la mémoire dans un état instable. Pour terminer un thread, il ne faut donc jamais utiliser l'une de ces méthodes.

2.6.2) Méthodes déconseillées

Pour les mêmes raisons que celles des méthodes `stop`, et parce qu'elles augmentent dans de trop grandes proportions la probabilité de placer le système dans une situation d'interblocage, la méthode `void suspend()` et la méthode `void resume()` sont déconseillées. Pour interrompre un thread, il ne faut donc jamais utiliser l'une de ces méthodes.

2.6.3) Exemple de terminaison correcte

Voici un exemple possible de démarrage et de terminaison correctes de thread, en utilisant une variable booléenne de contrôle. Ce n'est pas la seule manière de travailler, mais elle est assez propre :

```
private volatile boolean stopped;
private Thread thread;
void stopThread() {
    Thread moribund = thread;
    if (moribund != null) {
        stopped = true;
```

```
            // au cas où le thread serait bloqué
            moribund.interrupt();
            while (moribund.isAlive()) {
                try { moribund.join(); }
                catch (InterruptedException e) { /* rien */ }
            }
            thread = null;
        }
    }
    void startThread() {
        Thread emerging = thread;
        if (emerging == null) {
            emerging = new Thread(new Runnable() {
                public void run() {
                    stopped = false;
                    while (!stopped) {
                        // le thread travaille ici
                    }
                }
            });
            emerging.start();
            thread = emerging;
        }
    }
```

2.7) Priorité

La précédence d'un thread sur ses frères se mesure à l'aide de sa priorité donnée par `int getPriority()`. Elle vaut `NORM_PRIORITY` (5) par défaut. On peut la modifier avec `void setPriority(int)` où l'argument variera entre `MIN_PRIORITY` (1) et `MAX_PRIORITY` (10).

La priorité influence la fréquence d'attribution du CPU au thread par l'ordonnanceur (dans une certaine mesure, qui dépend du système hôte...).

2.8) Démons

Un démon est un thread qui tourne en tâche de fond au sein de l'application. La méthode `void setDaemon(boolean)` fait basculer l'état du thread entre thread démon et thread utilisateur. L'état démoniaque du thread peut être modifié autant de fois que nécessaire avant le démarrage ; une fois la méthode `start()` appelée, il ne peut plus être modifié.

La JVM s'arrête automatiquement lorsque les threads non démons se terminent ; les démons, eux, sont interrompus brutalement. Il est donc recommandé de ne pas laisser aux démons l'accès à des ressources comme les fichiers (perte de données en écriture).

3) Concurrence

3.1) Vocabulaire de la concurrence

3.1.1) Variables partagées

Les threads accèdent tous au même tas (celui de leur processus hôte) dans lequel ils peuvent lire et écrire des données par l'intermédiaire de variables dites partagées : c'est ainsi qu'ils communiquent entre eux. Nous rencontrerons alors des situations au cours desquelles plusieurs threads voudront accéder “en même temps” à une même variable partagée (accès concurrents). Pour que les modifications de la mémoire ne se produisent pas d'une manière qui serait nuisible à l'intégrité du système, on doit pouvoir garantir l'accès en exclusion mutuelle (par synchronisation).

Une variable locale (donc aussi un paramètre formel) ne peut pas être une variable partagée car elle n'existe que dans la pile d'exécution d'un thread, pas dans le tas. Les seules variables pouvant être partagées entre plusieurs threads sont donc :

- les variables d'instance
- les variables de classe
- les éléments d'un tableau

3.1.2) Situation de compétition (*race condition*)

Voici un exemple emblématique d’une situation de compétition : un thread s exécute le code :

```
resource = null;
```

pendant qu'un thread t exécute le code :

```
resource = new Resource();
if (resource != null) {
    System.out.println(resource.getData());
}
```

La variable `resource` est accédée à la fois par s et par t, en lecture et en écriture, et le résultat final dépend de l'ordre dans lequel ces deux threads y accèdent.

Nous dirons qu'il y a une situation de compétition (*race condition*) lorsque le comportement d’une portion de code dépend de l’ordre d’activation des threads mis en jeu.

Attention, il peut y avoir une situation de compétition, même pour l'exécution d'une instruction toute simple comme `i = i + 1;` !

3.1.3) Atomicité

Une opération est atomique relativement à une autre si son exécution ne peut pas être interrompue par celle de l'autre opération. Une opération est atomique si elle est atomique relativement à toute opération (y compris elle même) qui partage des variables avec elle. Évidemment, ces définitions sous-entendent que les opérations sont exécutées sur des

threads différents (lorsqu'il n'y a qu'un seul thread, toutes les opérations sont atomiques).

3.2) Variables volatiles

Lorsque deux threads accèdent concurremment à une même variable, il se peut que les modifications apportées par l'un ne soient pas visibles pour le second (cache, optimisation, lecture/écriture non atomique). Si l'on rend la variable `volatile`, Java garantit qu'elle sera toujours mise à jour dans, ou chargée depuis, la mémoire partagée lors de chaque utilisation.

3.3) Verrous et synchronisation

Chaque instance d'`Object` possède un verrou qu'il met à la disposition des threads pour leur permettre d'exécuter des portions de code en exclusion mutuelle. Les verrous n'interviennent pas dans l'exécution de code non synchronisé.

Lorsque le programmeur synchronise un bloc de code à l'aide du verrou d'un objet, juste avant que ce bloc soit exécuté sur le thread courant, ce dernier doit avoir obtenu le verrou de l'objet détenteur du bloc. Or ce verrou ne peut être pris que par un seul thread à la fois, ce qui en garantit l'utilisation en exclusion mutuelle. Après exécution du bloc synchronisé, le thread propriétaire du verrou le libère automatiquement. Un thread qui n'obtient pas un verrou est bloqué (état `BLOCKED`) ; il sera débloqué par le système lorsque le verrou sera relâché par le thread qui le détenait précédemment. Si plusieurs threads sont bloqués sur un même verrou, ce dernier est remis en jeu pour tous les threads à chaque fois, mais seul l'un d'entre eux sera le vainqueur, les autres seront de nouveau bloqués.

La notion de verrou permet la mise en œuvre de celle d'atomicité. Une méthode d'instance déclarée avec le mot clé `synchronized` est appelée normalement comme n'importe quelle autre méthode d'instance. Tout se passe comme si son corps était un bloc synchronisé sur le verrou de `this`. Une méthode statique déclarée avec le mot clé `synchronized` est appelée normalement comme n'importe quelle autre méthode statique. Tout se passe comme si son corps était un bloc synchronisé sur le verrou de la classe (vue en tant qu'objet à l'exécution).

Les verrous sont réentrants : un thread qui détient un verrou l'obtient automatiquement autant de fois que nécessaire. Le verrou compte le nombre de fois qu'il a été pris par un même thread ; il ne sera disponible que lorsque le thread qui le détenait l'aura rendu autant de fois qu'il l'avait obtenu.

Conservent les verrous précédemment acquis :

- tout thread évincé par l'ordonnanceur ;
- tout thread ayant rendu la main avec `yield` ;
- tout thread endormi avec `sleep`.

3.3.1) Méthodes wait et notify

Tout objet peut mettre un thread en attente ou le réveiller à l'aide des méthodes :

- `void wait()` met le thread courant en attente de notification de cet objet ;
- `void wait(long)` met le thread courant en attente de notification de cet objet

pendant une durée limitée (en ms) ;
- `void notify()` réveille un thread parmi ceux en attente de notification sur cet objet (choix aléatoire) ;
- `void notifyAll()` réveille tous les threads qui étaient en attente de notification sur cet objet.

Ces quatre méthodes nécessitent chacune d'être exécutées dans des blocs synchronisés sur l'objet cible de leur appel.

Au démarrage de `x.wait()` sur un thread `t`, si `t.isInterrupted()` alors la méthode s'arrête en levant immédiatement une `InterruptedException`. Sinon, `t` est placé dans la "salle d'attente" (*wait set*) de `x` par la JVM. Simultanément, `t` perd le verrou de `x` (autant de fois qu'il l'avait acquis) mais garde tous ses autres verrous. Plus tard, `t` devra récupérer le verrou de `x` pour pouvoir sortir de `wait`.

À l'appel de `x.notify()` sur un thread `s`, si la salle d'attente de `x` est non vide, un thread précédemment mis en attente par `x.wait()` est choisi au hasard puis réveillé. Supposons que ce soit le thread `t` dont il était question au paragraphe précédent. Mais `t` restera tout de même bloqué jusqu'à ce que `s` relâche effectivement le verrou de `x`, c'est-à-dire quand `s` sortira du bloc synchronisé qui encadre l'appel à `notify`. Il se pourrait aussi qu'un autre thread acquière le verrou de `x` avant que `t` ne puisse l'obtenir, cela retarderait d'autant la sortie de `t`...

Lorsque `t` acquiert enfin le verrou de `x`, la méthode `wait` se termine et `t` récupère le verrou de `x` autant de fois qu'il l'avait acquis avant d'entrer dans `wait`. En sortant du bloc synchronisé qui encadre l'appel à `wait`, `t` perdra une fois le verrou de `x`.

À l'appel de `x.notifyAll()` sur un thread `s`, la JVM éveille tous les threads précédemment mis en attente par `x.wait()`. Un seul d'entre eux obtiendra le verrou permettant de sortir, après que `s` l'aura relâché. Les autres seront eux aussi éveillés, mais bloqués jusqu'à l'obtention du verrou, chacun à leur tour... On ne peut rien dire sur l'ordre d'obtention. Chaque thread, après avoir obtenu le verrou de `x`, sort de `wait` puis relâche (une fois) le verrou.

3.3.2) Attente conditionnelle

Un thread peut vouloir attendre qu'une certaine condition soit remplie par un autre thread pour continuer à s'exécuter. Dans cette situation, on définit un objet servant de rendez-vous entre les deux threads. Lorsque le premier thread doit attendre que la condition soit remplie, il se met en attente sur l'objet rendez-vous. Lorsque le second thread est assuré que la condition est remplie, il notifie l'objet rendez-vous qui réveille le premier thread.

3.3.3) Spurious wakeup

Dans la documentation de la classe Thread, il est dit qu'un thread « *can wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied.* »

Par conséquent, il faut toujours s'assurer qu'un appel à `wait` est inclus dans une boucle qui va placer le thread en attente tant que la condition de poursuite n'est pas réalisée (et non pas *si* la condition n'est pas réalisée) et ceci, même s'il n'y a qu'un seul thread impliqué !

4) Thread safety

Une classe est dite thread-safe si chacune de ses instances respecte son contrat, même dans un environnement concurrent.

Un programme thread-safe est un programme qui respecte sa spécification, même dans un environnement concurrent.

L'API Java propose de telles classes à l'aide de méthodes synchronisées (`StringBuffer`, `Vector`, etc.) mais elle propose aussi des classes volontairement non thread-safe (`StringBuilder`, `Swing`, etc.) car ce n'est pas parce qu'une classe est thread-safe que tout programme qui l'utilise le sera aussi. En fait, un programme thread-safe peut très bien contenir des classes non thread-safe.

On donne cinq règles de thread safety :

> R1 : Toute classe sans état est thread-safe.
>
> R2 : Toute classe non mutable est thread-safe.
>
> R3 : Toute variable devant rester constante doit être déclarée **`final`** (en effet, lorsque plusieurs threads accèdent à un objet, java n'en garantit l'observation correcte, après exécution du constructeur, que pour ses attributs **`final`**).
>
> R4 : Toute variable partagée, à mutation progressive sur plusieurs threads, doit être gardée par un verrou (tout accès en lecture ou en écriture à cette variable doit se trouver synchronisé sur un seul et même verrou).
>
> R5 : Plusieurs variables partagées, liées par une même relation au sein d'un invariant, doivent être gardées par un même verrou et modifiées par des opérations atomiques.

5) Le modèle de mémoire Java

5.1) Visibilité de la mémoire

En java, les modifications de mémoire partagée par un thread ne sont généralement pas visibles depuis un autre thread. Tant que cela n'affecte pas la sémantique du code pour une exécution mono-thread, cela permet donc au compilateur de réordonner les instructions au niveau du pseudo-code et à la JVM de temporiser ou réordonner les opérations d'écriture en mémoire. Le code peut ainsi bénéficier d'optimisations agressives. Il y a donc toutes les chances que l'écriture d'un **`long`** (ou d'un **`double`**) se fasse à l'aide de deux opérations d'écriture de 32 bits, pas forcément consécutives. De plus, les valeurs des variables peuvent être stockées dans les registres et jamais réactualisées dans certains cas.

De fait, en présence de plusieurs threads, le code suivant :

```
class Test {
    private boolean ready;
    private int val;
    private long bigVal;
    void setData(int x) {
        val = x;
        bigVal = -x;
        ready = true;
    }
    void printData() {
        while (!ready) Thread.yield();
        System.out.println(val + " " + bigVal);
    }
}
```

assorti du programme :

```
final Test t = new Test();
new Thread(new Runnable() {
    public void run() {
        t.printData();
    }
}).start();
new Thread(new Runnable() {
    public void run() {
        t.setData(45);
    }
}).start();
```

peut théoriquement produire l'un quelconque des comportements suivants :

1. boucle indéfiniment
2. affiche "0 0"
3. affiche "0 4294967251"
4. affiche "0 -4294967296"
5. affiche "0 -45"
6. affiche "45 0"
7. affiche "45 4294967251"
8. affiche "45 -4294967296"
9. affiche "45 -45"

5.2) Le MMJ

Le langage Java spécifie son Modèle de Mémoire comme un jeu de règles décrivant les interactions entre les threads et la mémoire partagée, du point de vue de

- l'atomicité des actions : impossibilité de mixer l'exécution d'une action avec une

autre ;

- la visibilité des actions : possibilité pour un thread de se rendre compte des effets (sur la mémoire partagée) des actions effectuées par un autre thread ;
- l'ordonnancement des actions : description de l'ordre d'exécution des actions.

Le MMJ permet ainsi de déterminer, quelle que soit la situation, si une action effectuée par un thread peut, ou non, être prise en compte par un autre thread.

Les actions effectuées par les threads et décrites par le MMJ concernent

- la lecture ou l'écriture de variables partagées non volatiles ;
- les actions de synchronisation :
  - lecture ou écriture d'une variable partagée volatile ;
  - prise ou libération d'un verrou ;
  - démarrage ou observation de la mort d'un thread ;
  - première et dernière action (synthétiques) d'un thread ;
- les modifications de l'environnement d'exécution[6] ;
- les actions de divergence[7].

Pour simplifier, nous ne parlerons pas des deux dernières catégories.

D'une manière générale, la synchronisation coûte cher à mettre en place et l'on cherchera, quand c'est possible, à l'éviter. La connaissance du MMJ est indispensable pour nous permettre éventuellement de ne pas synchroniser.

Convention : sans nuire à la généralité du propos, on peut faire l'hypothèse que tout se passe comme si chaque thread était exécuté sur un processeur indépendant, interagissant avec sa mémoire cache et communiquant de temps en temps avec une mémoire partagée.

5.2.1) Garanties du MMJ en cas de synchronisation

Dans le cas d'une utilisation consistante de la synchronisation, le MMJ garantit que :

- toute action effectuée dans un bloc synchronisé est atomique par rapport à tout bloc synchronisé sur le même verrou ;
- toute action effectuée dans un bloc synchronisé est visible depuis tout bloc synchronisé sur le même verrou et exécuté ultérieurement[8] ;
- l'ordre des blocs synchronisés exécutés par un thread donné est cohérent avec l'ordre induit par la structure du code.

5.2.2) Atomicité minimale garantie par le MMJ

Tout accès en lecture ou en écriture à une variable partagée d'un type codé sur 32 bits ou moins est atomique. Pour garantir l'atomicité de ces opérations sur des variables partagées de type `long` ou `double`, il faut qu'elles soient volatiles. Attention, déclarer volatile une variable d'un type objet ne rend pas volatiles les champs de l'objet référencé.

6 Caractérisées par un affichage, une modification de fichier, etc.
7 Une action de divergence d'un thread est la modélisation du blocage infini de ce thread.
8 Le mot "ultérieurement" a bien un sens ici puisque les actions de synchronisation sont totalement ordonnées au sein d'une exécution (voir la section 5.2.5).

Même si, en lisant une variable non volatile, un thread est assuré d'obtenir une valeur cohérente (sa valeur initiale ou celle mise à jour sur un autre thread), il faut bien comprendre que ce ne sera pas forcément la dernière valeur mise à jour ! Mais il n'est pas impossible que ce soit la dernière valeur mise à jour... Attention donc à l'emploi de l'opérateur ++ (lecture + écriture !).

Comme on le sait déjà, toute action effectuée dans un bloc synchronisé *b* est atomique par rapport à toute action effectuée dans un bloc synchronisé sur le même verrou que *b*.

5.2.3) Visibilité minimale garantie par le MMJ

Toute action qui précède l'écriture d'une variable volatile est visible après une lecture de cette variable, subséquente à l'écriture.

Lorsqu'un thread libère un verrou, toute variable précédemment modifiée sur ce thread est mise à jour dans la mémoire partagée depuis le cache du thread. Lorsqu'un thread prend un verrou toute variable accessible sur ce thread est mise à jour dans le cache du thread depuis la mémoire partagée. Par conséquent, si un thread libère un verrou qui est ensuite pris par un autre thread, le second verra toutes les actions précédemment effectuées par le premier. Bien entendu, rien de tout cela n'est garanti si le verrou n'est pas commun aux deux threads.

5.2.4) Ordre minimal garanti par le MMJ

Du point de vue du thread sur lequel s'exécute le code d'une méthode, les instructions se déroulent dans l'ordre du code cible, mais du point de vue des autres threads qui observent le précédent à travers l'exécution de code non synchronisé ou de variables non volatiles, n'importe quel ordre peut être perçu. Mais l'ordre d'exécution relatif des blocs synchronisés ou des opérations sur les variables volatiles est préservé.

Attention toutefois, il est très fréquent que les autres threads observent le “bon” ordre !

5.2.5) Synthèse et compléments

Les quatre sous-sections précédentes n'expriment pas encore la totalité des règles du MMJ. C'est dans cette sous-section que nous allons le faire.

Formellement, le MMJ définit une relation d'antériorité (*happens-before*) qui induit un ordre partiel sur les actions des threads[9]. Cette relation est définie ainsi :

- pour un même thread `t`, une action `x` est antérieure à une action `y` si `x` précède `y` dans la séquence d'instructions du code cible de `t` ;
- la libération d'un verrou est antérieure à toute future acquisition de ce verrou ;
- l'écriture d'une variable volatile est antérieure à toute future lecture de la variable ;
- un appel `t.start()` est antérieur à toute action de `t` ;
- toute action de `t` est antérieure au retour de `t.join()` ou au retour de la valeur **`false`** par `t.isAlive()` ;
- une interruption est antérieure à l'observation de cette interruption (levée d'exception, lecture du drapeau d'interruption) ;

9 Néanmoins, si l'on se restreint aux actions de synchronisation, cette relation induit un ordre total.

- l'initialisation standard d'un objet est antérieure à tout appel à `start` ;
- la fin de l'exécution du constructeur d'un objet est antérieure au début de l'exécution de son finaliseur ;
- la relation d'antériorité est transitive.

Le MMJ garantit que « si une action `x` est antérieure à une action `y`, alors l'effet de `x` est visible depuis `y`, et `x` est ordonnée avant `y` ». Dans tous les cas où `x` n'est pas antérieure à `y`, dans le but de permettre une amélioration de l'efficacité du code, le compilateur et la JVM sont libres de réordonner ces actions comme ils le souhaitent.

5.3) Un exemple complet : le musée

Un musée présente une exposition perpétuelle. L'exposition est retransmise en direct par un cameraman qui filme, continuellement et en plan fixe, les tableaux de l'exposition. Le directeur du musée change de temps en temps le style de son exposition en y plaçant des tableaux d'un nouveau peintre. Le cameraman ne doit pas transmettre durant le changement des tableaux... voir en cours.

On modifie ensuite légèrement l'énoncé précédent : on ne dispose plus d'un cameraman, mais d'un photographe ; il ne sert donc à rien de prendre plusieurs fois la même photo de l'exposition et, entre deux photos, un photographe attendra que le directeur ait changé l'exposition... voir en cours.

On modifie encore un peu plus l'énoncé : on dispose maintenant de plusieurs photographes ; la pellicule photo coûtant cher, si un photographe a déjà pris la photo, les autres, ne voulant pas gaspiller leur pellicule, ne prendront pas de photo cette fois-ci et un seul photographe sera publié... voir en cours.

6) Collections

6.1) Collections synchronisées

Ce sont des collections accessibles à partir de méthodes statiques de la bibliothèque `java.util.Collections` et d'une collection non synchronisée :

- `Collection<E> synchronizedCollection(Collection<E>)`
- `Set<E> synchronizedSet(Set<E>)`
- `List<E> synchronizedList(List<E>)`
- `Map<K, V> synchronizedMap(Map<K, V>)`
- `SortedSet<E> synchronizedSortedSet(SortedSet<E>)`
- `SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V>)`

Les instances de collections que l'on obtient à partir de ces méthodes sont des *proxy* sur les collections fournies en paramètre, dont les méthodes sont toutes synchronisées sur un *mutex* interne.

6.2) Collections thread-safe

Les collections de `java.util.concurrent` sont thread-safe, mais elles n'utilisent pas (ou très peu) la synchronisation. Du coup, elles sont plus efficaces et plus facilement « parallélisables » (car la synchronisation entraîne la séquentialisation). Mais elles ne peuvent pas être utilisées de manière atomique car aucun verrou ne garantit l'utilisation exclusive des SdD internes de ces collections. Néanmoins, certaines méthodes atomiques sont prévues (comme par exemple `putIfAbsent` de `ConcurrentMap`).

6.2.1) Implantations de List

- `CopyOnWriteArrayList` : implantation thread-safe par tableaux non mutables
  - chaque modification crée un nouveau tableau
  - toute lecture est constante
  - intéressant si peu de modifications et beaucoup de lectures
  - se synchronise de manière interne uniquement lors de la création d'un nouveau tableau
  - aucune synchronisation externe nécessaire, même pour l'itération
  - itérateurs *snapshot* (travaille sur une copie des données, créée lors de l'instanciation de l'itérateur)
  - pas de `ConcurrentModificationException`
  - les modifications à partir de l'itérateur ne sont pas supportées

6.2.2) Implantations de Set

- `CopyOnWriteArraySet` (implantation de `Set`) : *idem* `CopyOnWriteArrayList`
- `ConcurrentSkipListSet` (implantation de `NavigableSet`) : implantation à base de listes à enjambements (*skip list*)
  - performances comparables aux arbres binaires équilibrés (insertion/recherche en O(log n))
  - performance optimale (constante) lors de l'itération
  - seule opération synchronisée : `size` (mais coûte cher)
  - pas de synchronisation pour insertion/suppression
  - itérateurs *weakly consistent* (aucune garantie sur la prise en compte ou non des modifications concurrentes)
  - pas de `ConcurrentModificationException`
  - supporte les modifications à partir de l'itérateur

6.2.3) Implantations de Map

- `ConcurrentHashMap` (implantation de `Map`)
  - pas de synchronisation pour la recherche
  - table segmentée permettant des mises-à-jour concurrentes non synchronisées
  - aussi efficace qu'une `HashMap` classique
  - Seule opération synchronisée : `size` (mais coûte cher)
  - itérateurs *weakly consistent*

- `ConcurrentSkipListMap` (implantation de `NavigableMap`) : *idem* `ConcurrentSkipListSet`

6.2.4) Implantations de Queue (non bloquante)

- `ConcurrentLinkedQueue`
  - utilise les instructions CAS (*Compare And Swap*) des systèmes multi-cœurs
  - insertions et suppressions concurrentes en temps constant
  - seule opération synchronisée : `size` (mais coûte cher : O(n))
  - itérateurs *weakly consistent*

Pour rappel, les méthodes de l'interface `Queue` peuvent échouer de deux façons :

- par levée d'exception
  - `boolean add(E)`
  - `E remove()`
  - `E element()`
- par retour de valeur spéciale (`null` ou `false`)
  - `boolean offer(E)`
  - `E poll()`
  - `E peek()`

6.2.5) Implantations de Queue (bloquante)

Ces SdD notifient les threads quand elles sont prêtes :

- `ArrayBlockingQueue` : file bornée
- `LinkedBlockingQueue` : file bornée ou non bornée
- `PriorityBlockingQueue` : file de priorité
- `DelayQueue` : file de priorité non bornée (les éléments sont des instances de l'interface `Delayed`)
- `SynchronousQueue` : fausse file permettant de synchroniser les threads

Pour rappel, les méthodes de `BlockingQueue` peuvent attendre de deux façons :

- indéfiniment
  - `void put(E)`
  - `E take()`
- pendant un certain temps
  - `boolean offer(E, long, TimeUnit)`
  - `E poll(long, TimeUnit)`

7) Swing et les threads

On le sait bien, Swing n'est pas thread-safe. Un seul thread est autorisé à accéder aux composants graphiques, ce qui permet d'éviter les situations d'interblocage et les problèmes de partage de la mémoire. EDT est ce thread, seul habilité à manipuler des composants Swing ; il doit effectuer toutes les opérations de dessin (`PaintEvent`) et exécuter toutes les méthodes réflexes de tous les écouteurs des composants graphiques.

Par conséquent, une méthode réflexe doit être d'exécution rapide sinon elle doit déléguer (une partie de) son exécution à un autre thread. Si cet autre thread veut manipuler des composants Swing, il doit demander à EDT de le faire, de manière asynchrone : `void {SwingUtilities|EventQueue}.invokeLater(Runnable)` ou de manière synchrone : `void {SwingUtilities|EventQueue}.invokeAndWait(Runnable)`. Pour s'assurer que les `PropertyChangeListener` (`java.beans`) seront bien notifiés sur EDT, on peut utiliser `SwingPropertyChangeSupport` (`javax.swing.event`).

On a pu croire un temps que « un composant Swing peut être manipulé par n'importe quel thread tant qu'il n'est pas réalisé », c'est-à-dire qu'un composant Swing peut être créé dans un thread puis manipulé ensuite avec EDT. C'est faux[10] !

Quelques méthodes sont thread-safe dans Swing, par exemple

- dans `Component`
  - `void repaint()`
  - `void revalidate()`
- dans `Container`
  - `<T extends EventListener> T[] getListeners(Class<T>)`

ainsi que toutes les méthodes

- `void addXListener(XListener)`
- `void removeXListener(XListener)`

Attention ! Dans la doc Java 6 la méthode `append` de `JTextArea` est soit-disant thread-safe... c'est vrai dans un sens : `insertString` de `AbstractDocument` utilise un verrou, mais cette opération n'est pas atomique ! D'ailleurs, la doc Java 7 ne prétend plus que cette méthode est thread-safe ! Conclusion : faites comme si `append` n'était pas thread-safe.

7.1.1) javax.swing.Timer

Dans certains cas (typiquement les animations) on peut découper la tâche de départ en tâches cycliques suffisamment courtes. Par exemple l'animation d'un objet à l'écran peut être découpée en mouvements élémentaires exécutés à intervalles réguliers.

On peut alors utiliser un `javax.swing.Timer` qui va générer des `ActionEvent` à intervalles réguliers, ces derniers étant écoutés par un `ActionListener` (dont la méthode `actionPerformed` est d'exécution garantie sur EDT).

10 Lire par exemple `http://web.archive.org/web/20090925191745/http://eppleton.sharedhost.de/blog/?p=806`

Toutes les x ms (défini à la construction), un `ActionEvent` est généré par le timer et EDT exécute la méthode réflexe de l'`ActionListener` donné au constructeur pour traiter cet événement. Si le code de la méthode réflexe est court, il ne fige pas l'application ; de plus, il doit contenir sa propre condition d'arrêt.

On stoppe le timer avec `stop()` et on peut le redémarrer à tout moment en appelant `restart()`.

7.1.2) javax.swing.SwingWorker

Cette nouvelle classe (Java SE 6) permet d'exécuter un long traitement en tâche de fond (méthode `doInBackground`) sur un thread interne dédié, d'annuler le traitement en cours (méthode `cancel`), d'exécuter une action complémentaire sur EDT après la fin du traitement (méthode `done`), de communiquer avec EDT en cours de traitement en publiant des résultats intermédiaires (méthode `publish`) traités par EDT (méthode `process`), et de notifier les changements des propriétés liées `state` (une constante de `SwingWorker.StateValue` parmi `PENDING`, `STARTED`, `DONE`) et `progress` (un entier entre 0 et 100).

`SwingWorker<T, V>` est une classe abstraite générique, `T` est le type de la valeur retournée en fin de traitement par

- **`protected abstract`** `T doInBackground()`
- **`public`** `T get()` (méthode bloquante, attendant la fin d'exécution de `doInBackground` pour retourner sa valeur)

et `V` est le type des valeurs communiquées entre le worker et l'EDT en cours de traitement avec

- **`protected void`** `publish(V...)` : envoie un nombre quelconque de valeurs intermédiaires à la méthode `process` qui les traitera depuis EDT (ne doit être utilisée que dans `doInBackground`) ;
- **`protected void`** `process(List<V>)` traite un nombre quelconque de valeurs (est exécutée sur EDT).

Rq. : il peut y avoir plusieurs appels à `publish` avant qu'un `process` ne s'exécute (la liste correspond alors à la concaténation des tableaux).

On notera aussi que l'absence de valeur se traduit par le type `Void` qui n'admet aucune instance (seule valeur possible : **`null`**).

Chapitre 4 : composant JList

1) Généralités

Le composant `javax.swing.JList` permet l'affichage d'un groupe d'éléments en autorisant la sélection simple ou multiple (contiguë ou non) parmi ces éléments.

Ce composant graphique Swing est doté de deux modèles : un `ListModel` qui gère les éléments que la liste doit afficher et un `ListSelectionModel` qui gère le mécanisme de sélection de ces éléments par l'utilisateur.

Il est aussi doté d'une partie graphique qui agrège la vue et le contrôleur sous la forme d'un seul composant de type `ListUI`. Ce dernier observe les deux modèles précédents à l'aide d'un `ListDataListener`, pour le premier, et d'un `ListSelectionListener` pour le second. Mais il écoute aussi les actions de l'utilisateur à l'aide d'un `KeyListener`, d'un `MouseInputListener` et d'un `FocusListener` pour pouvoir traduire les actions de l'utilisateur en commandes à destination des différents modèles.

1.1) Configuration

Il dispose, entre autres propriétés intéressantes, des deux propriétés suivantes :

- `layoutOrientation` : propriété liée, de type **`int`**, pouvant prendre une valeur parmi les constantes suivantes de `JList`
  - `HORIZONTAL_WRAP` : disposition horizontale des éléments avec passage automatique à la ligne suivante,
  - `VERTICAL_WRAP` : disposition verticale des éléments avec passage automatique à la colonne suivante,
  - `VERTICAL` : disposition des éléments sur une seule colonne.
- `selectionMode` : propriété non liée, de type **`int`**, pouvant prendre une valeur parmi les constantes suivantes de `ListSelectionModel`
  - `SINGLE_SELECTION` : on ne peut sélectionner qu'un seul élément à la fois,
  - `SINGLE_INTERVAL_SELECTION` : on peut sélectionner plusieurs éléments contigus à la fois,
  - `MULTIPLE_INTERVAL_SELECTION` : on peut sélectionner plusieurs éléments à la fois, même s'ils ne sont pas contigus.

1.2) Mise en place

Les clients disposent de quatre constructeurs :

- `JList()` : une liste de modèle vide en lecture seule ;
- `JList(Object[] listData)` : une liste de modèle initialisé avec `listData` en lecture seule ;
- `JList(Vector<?> listData)` : une liste de modèle initialisé avec `listData` en lecture seule ;

- `JList(ListModel listData)` : une liste de modèle (non **`null`**) `listData`.

2) Modèle de données

Le composant est doté d'une propriété liée `model` de type `ListModel`, accessible en lecture/écriture :

- `ListModel getModel()`
- **`void`** `setModel(ListModel)`

Notez bien que les méthodes suivantes :

- **`void`** `setListData(Object[])`
- **`void`** `setListData(Vector<?>)`

créent un modèle accessible en lecture seulement, dont les éléments sont donnés par le paramètre.

Tout changement de modèle est notifié aux `PropertyChangeListener`s enregistrés sur la propriété `model`. Il faut savoir aussi que l'on ne peut modifier les données d'une liste (changement, ajout ou retrait d'élément) qu'à partir de son modèle et non directement à l'aide du composant graphique.

2.1) ListModel : définition des modèles de listes

Cette interface définit les modèles de listes à l'aide de quatre méthodes :

- **`void`** `addListDataListener(LDL)` : associe un `LDL` ;
- **`void`** `removeListDataListener(LDL)` : désassocie un `LDL` ;
- **`int`** `getSize()` : le nombre d'éléments dans le modèle ;
- `Object getElementAt(`**`int`**`)` : le i-ème élément du modèle.

La documentation explique qu'un modèle de liste doit fonctionner comme un vecteur dont les indices varient de 0 à `getSize()` - 1. Tout changement de contenu ou de longueur dans le modèle doit être notifié aux `ListDataListener`s préalablement enregistrés, sous peine d'empêcher le composant de se rafraîchir, car son *UI-delegate* utilise de tels écouteurs pour son fonctionnement interne.

2.2) AbstractListModel : implantation partielle des modèles de listes

Dans cette classe abstraite, on trouve sept méthodes concrètes :

- **`void`** `addListDataListener(LDL)`
- **`void`** `removeListDataListener(LDL)`
- **`protected void`** `fireIntervalAdded(Object, int, int)`
- **`protected void`** `fireIntervalRemoved(Object, int, int)`
- **`protected void`** `fireContentsChanged(Object, int, int)`
- `<T` **`extends`** `EventListener> T[] getListeners(Class<T>)`
- `ListDataListener[] getListDataListeners()`

et seulement deux méthodes abstraites :

- **`int`** `getSize()`

- `Object getElementAt(int)`

Le mécanisme de notification est préimplanté à travers les trois méthodes concrètes protégées `fire*`. On notera l'absence de méthode prenant en charge la modification du modèle...

2.3) Implantations des modèles de listes

Les modèles sous-jacents aux trois constructeurs `JList()`, `JList(Object[])` et `JList(Vector<?>)` sont des implantations triviales et non mutables de `AbstractListModel`. Elles sont réalisées sur le principe suivant :

```
public JList(final Object[] listData) {
    this(new AbstractListModel() {
        public int getSize() {
            return listData.length;
        }
        public Object getElementAt(int i) {
            return listData[i];
        }
    });
}
```

Si l'on veut manipuler des modèles mutables, on peut utiliser la classe `DefaultListModel` ou définir des modèles personnalisés.

Toute classe de gestion des données d'une liste, qu'elle soit mutable ou non mutable, doit obligatoirement s'assurer :

- d'implanter l'interface `ListModel`
- de gérer les notifications aux `ListDataListeners` (car le délégué `ListUI` enregistre un tel écouteur pour effectuer les mises à jour d'affichage)
- et éventuellement dériver `AbstractListModel` ou `DefaultListModel` (c'est conseillé car la gestion des `ListDataEvent` est facilitée par la présence des méthodes `fire*`).

2.4) ListDataListener

Cette interface d'écouteurs définit trois méthodes :

- `void contentsChanged(ListDataEvent e)` : activée lorsque les éléments du modèle situés entre les indices `e.getIndex0()` et `e.getIndex1()` (inclus) ont été modifiés
- `void intervalAdded(ListDataEvent e)` : activée lorsque les éléments du modèle situés entre les indices `e.getIndex0()` et `e.getIndex1()` (inclus) viennent d'être insérés dans le modèle
- `void intervalRemoved(ListDataEvent e)` : activée lorsque les éléments du modèle anciennement situés entre les indices `e.getIndex0()` et `e.getIndex1()` (inclus) ont été supprimés du modèle.

3) Modèle de sélection des éléments

Les sélections (par simple clics) sont gérées par un modèle dédié qui implante `ListSelectionModel`. À l'inverse de la gestion des données, toutes les méthodes de `ListSelectionModel` sont intégrées au composant de sorte que l'on peut accéder aux outils de sélection directement à partir d'une instance (qui délègue, bien entendu, le traitement à son modèle de sélection) :

- `void setSelectionInterval(int index0, int index1)` : sélectionne les éléments entre les positions `index0` et `index1` (incluses)
- `void clearSelection()` : désélectionne la sélection courante
- `void addSelectionInterval(int index0, int index1)` : ajoute à la sélection courante les éléments entre les positions `index0` et `index1` (incluses)
- `void removeSelectionInterval(int index0, int index1)` : supprime de la sélection courante les éléments entre les positions `index0` et `index1` (incluses).

Les propriétés des `JLists` en rapport avec la sélection sont au nombre de douze :

- `anchorSelectionIndex` (**`int`**, R) : l'index de début de sélection lors de la dernière opération
- `leadSelectionIndex` (**`int`**, R) : l'index de fin de sélection lors de la dernière opération
- `maxSelectionIndex` (**`int`**, R) : la plus grande valeur d'index lors de la dernière opération
- `minSelectionIndex` (**`int`**, R) : la plus petite valeur d'index lors de la dernière opération
- `selectedIndex` (**`int`**, RW) : le plus petit index actuellement sélectionné
- `selectedIndices` (**`int[]`**, RW) : l'ensemble des valeurs d'index actuellement sélectionnés
- `selectedValue` (`Object`, R) : le premier élément actuellement sélectionné
- `selectedValues` (`Object[]`, R) : l'ensemble des éléments actuellement sélectionnés
- `selectionEmpty` (**`boolean`**, R) : indique si la sélection est vide
- `selectionMode` (**`int`**, RW) : le mode de sélection en cours
- `selectionModel` (`ListSelectionModel`, RW, liée) : le modèle de sélection en cours
- `valueIsAdjusting` (`boolean`, RW) : indique si l'opération actuelle de sélection multiple est en cours ou est terminée.

3.1) ListSelectionEvent

Un `ListSelectionEvent` est créé pour chaque action de sélection effectuée soit par le client humain (souris/clavier), soit par le programmeur (opérations de sélection).

Les méthodes de `ListSelectionEvent` sont les suivantes :

- `Object getSource()` : la `JList` ou son `ListSelectionModel` selon le cas

- `int getFirstIndex()` : l'index du premier élément dont la valeur de sélection a changé
- `int getLastIndex()` : l'index du dernier élément dont la valeur de sélection a changé
- `boolean getValueIsAdjusting()` : retourne `true` au début d'une opération de sélection, `false` dès qu'elle est terminée.

3.2) ListSelectionListener

Un `ListSelectionListener` peut être enregistré auprès d'une `JList` (recommandé) ou bien directement auprès de son `ListSelectionModel` ; la différence de fonctionnement se résume à la différence dans la source des événements.

La méthode réflexe d'un `ListSelectionListener` possède le prototype suivant : `void valueChanged(ListSelectionEvent)`.

3.3) Clics multiples

Les `JList`s ne gèrent que le simple clic, avec appui simultané sur les touches Shift ou Ctrl pour les sélections multiples. Pour prendre en compte les clics multiples il faut utiliser un `MouseListener`...

4) Rendu graphique des éléments

4.1) Défilement des éléments

Les `JList`s ne gèrent pas elles-mêmes le défilement de leurs éléments. Pour cela il faut les disposer sur un `JScrollPane` (ce qui est possible puisqu'elles sont `Scrollable`). Néanmoins, certaines des propriétés de JList sont en rapport avec le défilement :

- `firstVisibleIndex` (`int`, R) : index de la première ligne visible
- `lastVisibleIndex` (`int`, R) : index de la dernière ligne visible
- `preferredScrollableViewportSize` (`Dimension`, R) : dimension du viewport nécessaire pour voir `visibleRowCount` lignes
- `visibleRowCount` (`int`, RW, liée) : nombre de lignes à afficher

et la méthode `void ensureIndexIsVisible(int)` permet de s'assurer qu'une ligne particulière est bien visible.

4.2) ListCellRenderer

Une `JList` affiche ses éléments au moyen d'un *renderer*. Il s'agit d'un composant graphique qui implante l'interface `ListCellRenderer` et qui sait afficher les éléments constitutifs d'une liste quelle que soit leur nature. Par défaut, un *renderer* sait afficher les images et les chaînes de caractères ; pour les autres types de composants, il utilise leur méthode `toString`.

Bien qu'elle ait plusieurs éléments à afficher, une `JList` n'utilise qu'un seul *renderer* : à chaque fois qu'elle doit afficher l'un de ses éléments, elle procède en trois étapes

1. elle configure le *renderer* avec `renderer.getListCellRendererComponent`
2. elle positionne le *renderer* avec `renderer.setBounds`
3. elle dessine le *renderer* avec `renderer.paint`.

C'est bien la même instance de *renderer* qui est utilisée autant de fois qu'il y a d'éléments de la `JList` à dessiner.

4.2.1) Définition d'un renderer

Il faut obligatoirement implanter l'interface `ListCellRenderer`, dotée d'une seule méthode :

```
Component getListCellRendererComponent(
    JList list,   // la liste cliente
    Object value, // l'élément à afficher
    int index,    // la position de l'élément
    boolean isSelected,  // l'élément est-il sélectionné ?
    boolean cellHasFocus // l'élément a-t-il le focus ?
)
```

Attention : cette méthode ne doit pas créer un nouveau composant mais plutôt adapter **`this`** ou utiliser un délégué. Si nécessaire, il faudra adapter la méthode `paintComponent` du composant retourné... voir TP.

La classe fournie par l'API, `DefaultListCellRenderer`, implante `ListCellRenderer` et étend `JLabel` :

- on dérive `JLabel` pour réutiliser la faculté des labels à afficher les images et les chaînes
- on redéfinit certaines méthodes pour optimiser l'affichage du label dans ce cas particulier.

La méthode `getListCellRendererComponent` définit l'orientation du composant ainsi que ses couleurs de fond et de texte en fonction des valeurs correspondantes de la liste et du paramètre `isSelected`. Elle dessine une image ou une chaîne selon le type du paramètre `value` et configure la bordure selon la valeur des paramètres `isSelected` et `cellHasFocus`, et en fonction du *Look&Feel*. Enfin, elle retourne **`this`**.

La méthode `paintComponent` n'est pas modifiée. Toutes les méthodes [`in`|`re`]`validate` et `repaint` sont dotées d'un corps vide car le *renderer* est uniquement dessiné par appel direct à `paint`. Seule la méthode `firePropertyChange` pour des valeurs de type `Object` n'est pas vide, pour permettre de notifier les changements des propriétés liées `text`, `font` et `foreground` (uniquement). Les autres méthodes `firePropertyChange` sont dotées d'un corps vide.

Chapitre 5 : composant JTable

`javax.swing.JTable` est un composant graphique permettant l'affichage et l'édition de données sous forme tabulaire (comme dans une feuille de calcul d'un tableur). Chaque élément de donnée est affiché dans une zone appelée cellule. Les cellules sont regroupées horizontalement pour former des lignes, et verticalement pour former des colonnes. Dans une `JTable`, toutes les cellules d'une même colonne doivent être du même type. Une colonne peut être dotée d'un entête affichant une information sur le type des cellules qu'elle contient.

`JTable` utilise un grand nombre de classes et d'interfaces, situées pour la plupart dans le paquetage `javax.swing.table`. La gestion des données stockées dans les cellules est confiée à une instance de `TableModel`, mais la gestion des données par colonne (au niveau du modèle, pas au sens des colonnes graphiques) est réservée aux instances de `TableColumnModel` et de `TableColumn`. Les entêtes de colonnes sont des instances de `JTableHeader`. La sélection des cellules est gérée par un `ListSelectionModel`, tandis que le rendu visuel des cellules se fait par l'intermédiaire d'un `TableCellRenderer` et l'édition des cellules par un `TableCellEditor`.

1) Mise en place

- `JTable()` : table avec zéro cellule
- `JTable(int r, int c)` : table avec `r` × `c` cellules vides
- `JTable(Object[][] rowCells, Object[] colNames)` : table définie par un tableau de tableaux de données et un tableau d'entêtes de colonnes
- `JTable(Vector rowCells, Vector colNams)` : table définie par un vecteur de vecteurs de données
- `JTable(TableModel tm)` : table définie par un modèle de données quelconque
- `JTable(TableModel tm, TableColumnModel tcm)` : table définie par un modèle de données et un modèle de colonnes quelconques
- `JTable(TableModel tm, TableColumnModel tcm, ListSelectionModel lsm)` : table définie par un modèle de données, un modèle de colonnes et modèle de sélection quelconques.

2) Modèle de données

Les données d'une `JTable` sont spécifiées par l'interface `TableModel` :

- `void addTableModelListener(TML)` : ajoute un écouteur de modification des données
- `void removeTableModelListener(TML)` : retire un écouteur de modification des données
- `int getColumnCount()` : le nombre de colonnes du modèle
- `int getRowCount()` : le nombre de lignes du modèle

- `String getColumnName(int)` : le nom associé à une colonne
- `Class<?> getColumnClass(int)` : la classe commune aux objets d'une colonne
- `Object getValueAt(int, int)` : donne la valeur d'une cellule
- `void setValueAt(Object, int, int)` : fixe la valeur d'une cellule
- `boolean isCellEditable(int, int)` : indique si une cellule est éditable

2.1) Notifications de changement d'état du modèle

L'interface `TableModelListener` ne possède qu'une seule méthode réflexe : `void tableChanged(TableModelEvent)`.

2.1.1) Description des TableModelEvent

- `int getColumn()` : si égal à `ALL_COLUMNS`, alors toutes les colonnes ont changé, sinon c'est l'indice de la colonne modifiée
- `int getFirstRow()` : si égal à `HEADER_ROW`, alors c'est l'en-tête qui a changé, sinon c'est l'indice de la première ligne modifiée
- `int getLastRow()` : indice de la dernière ligne modifiée
- `int getType()` : `INSERT` ou `UPDATE` ou `DELETE`

2.1.2) Constructeurs de TableModelEvent

Paramètres des constructeurs :

- `TableModel src` : n'importe quelle cellule peut avoir changé
- `TableModel src`, `int row` : seules les cellules de la ligne `row` peuvent avoir changé ; si `row == HEADER_ROW` alors il s'agit de la ligne d'entête
- `TableModel src`, `int firstRow`, `int lastRow` : seules les cellules entre les lignes `firstRow` et `lastRow` peuvent avoir changé
- `TableModel src`, `int firstRow`, `int lastRow`, `int col` : seules les cellules entre les lignes `firstRow` et `lastRow` en colonne `col` peuvent avoir changé ; si `col == ALL_COLUMN` alors toutes les colonnes sont concernées
- `TableModel src`, `int firstRow`, `int lastRow`, `int col`, `int type` : *idem* ci-dessus avec précision sur le type de modification

2.1.3) Détail des méthodes fire* de AbstractTableModel

C'est au modèle de s'assurer que les écouteurs seront bien notifiés des modifications. La méthode de base est `void fireTableChanged(TableModelEvent)`, c'est elle qui distribue l'événement à tous les écouteurs enregistrés.

Pour faciliter la vie du programmeur, `AbstractTableModel` met plusieurs autres méthodes à sa disposition. Chacune appelle `fireTableChanged` mais pour des `TableModelEvent` construits différemment :

- `fireTableDataChanged()` : génère `new TableModelEvent(this)`
- `fireTableCellUpdated(int row, int column)` : génère `new TableModelEvent(this, row, row, column)`
- `fireTableRowsDeleted(int firstRow, int lastRow)` : génère `new TableModelEvent(this, firstRow, lastRow, ALL_COLUMNS, DELETED)`
- `fireTableRowsInserted(int firstRow, int lastRow)` : génère `new TableModelEvent(this, firstRow, lastRow, ALL_COLUMNS, INSERTED)`
- `fireTableRowsUpdated(int firstRow, int lastRow)` : génère `new TableModelEvent(this, firstRow, lastRow, ALL_COLUMNS, UPDATED)`
- `fireTableStructureChanged()` : génère `new TableModelEvent(this, HEADER_ROW)`

2.2) Implantation partielle du modèle

La classe `AbstractTableModel` est une implantation partielle du modèle de données. Elle concrétise les six méthodes :

- `getColumnName` : retourne 'A', 'B', etc.
- `getColumnClass` : retourne `Object.class`
- `isCellEditable` : retourne `false`
- `setValueAt` : ne fait rien
- `addTableModelListener` : gère la liste d'écouteurs du modèle de données
- `removeTableModelListener` : gère la liste d'écouteurs du modèle de données

et conserve trois méthodes abstraites :

- `getRowCount`
- `getColumnCount`
- `getValueAt`

2.2.1) Exemple de modèle non éditable

```
class SimpleROTableModel extends AbstractTableModel {
    private Object[][] data;
    public SimpleROTableModel(Object[][] data) {
        if (data == null) {
            throw new IllegalArgumentException();
        }
        this.data = data;
    }
    public int getRowCount() {
        return data.length;
    }
    public int getColumnCount() {
```

```
        return data.length == 0 ? 0 : data[0].length;
    }
    public Object getValueAt(int row, int col) {
        return data[row][col];
    }
}
```

2.2.2) Exemple de modèle éditable :

```
class SimpleRWTableModel extends AbstractTableModel {
  private Object[][] data;
  private Object[] colNames;
  public SimpleRWTableModel(Object[][] data, Object[] colNames) {
    // ...
    this.data = data;
    this.colNames = colNames;
  }
  public int getRowCount() {
    return data.length;
  }
  public int getColumnCount() {
    return colNames.length;
  }
  public Object getValueAt(int row, int col) {
    return data[row][col];
  }
  @Override public String getColumnName(int column) {
    return colNames[column].toString();
  }
  @Override public boolean isCellEditable(int row, int column) {
    return true;
  }
  @Override public void setValueAt(Object value, int r, int c) {
    data[r][c] = value;
    fireTableCellUpdated(r, c);
  }
}
```

2.3) Implantation standard

La classe `DefaultTableModel` implante un modèle de données standard à l'aide d'un `Vector` de `Vector`s. Il faut agir prudemment car toutes les données sont simultanément présentes en mémoire ! Si les données sont très nombreuses (base de données, fichier volumineux), cette implémentation n'est donc pas adaptée. Dans ces cas de figure, on préférera adapter `AbstractTableModel` par dérivation.

2.3.1) Remplissage d'un modèle standard

- `void addColumn(Object)` : ajoute une colonne (éventuellement vide) et son nom au modèle, en dernière position

- `void addRow(Object[])` : ajoute une ligne au modèle
- `void insertRow(int, Object[])` : insère une ligne dans le modèle
- `void setValueAt(Object, int, int)` : modifie une cellule du modèle
- `void setDataVector(Object[][], Object[])` : change la structure complète du modèle (cellules et noms des colonnes)

Les quatre dernières méthodes existent aussi avec des paramètres de type `Vector` au lieu des tableaux d'objets...

2.4) TableColumn

`TableColumn` gère tous les attributs des colonnes affichées : tailles, rendu graphique, éditeur de cellules et position dans le modèle. Les principales propriétés de ce composant sont :

- `headerValue` (`Object`, RW, liée)
- `identifier` (`Object`, RW, liée)
- `maxWidth` (**`int`**, RW, liée)
- `minWidth` (**`int`**, RW, liée)
- `modelIndex` (**`int`**, RW, liée)
- `preferredWidth` (**`int`**, RW, liée)
- `resizable` (**`boolean`**, RW, liée)
- `width` (**`int`**, RW, liée)

2.4.1) Modèles de colonnes

`JTable` délègue la gestion de l'affichage des colonnes à un modèle de type `TableColumnModel` qui gère les instances de `TableColumn`, la sélection des colonnes, le déplacement des colonnes, la correspondance entre la position de chaque colonne dans la `JTable` et celle qui lui correspond dans le modèle.

`DefaultTableColumnModel` est une implantation standard de l'interface `TableColumnModel`.

À la création d'une `JTable`, lors du changement de modèle de données ou lors d'un événement indiquant un changement de la structure complète de la table, un `DefaultTableColumnModel` est automatiquement créé sur la base des cellules présentes dans le modèle. On peut contrôler ce comportement par défaut avec la propriété `autoCreateColumnsFromModel` (**`boolean`**, RW, liée) des `JTable`s.

Le modèle de colonnes peut ne pas refléter exactement le modèle de données. Par exemple : une table avec affichage en double des colonnes du modèle de données...

On peut aussi choisir de ne pas afficher certaines colonnes : il suffit de retirer les `TableColumns` représentant ces colonnes du `TableColumnModel`.

2.4.2) TableColumnModelListener

Ces écouteurs sont notifiés de tout ajout, retrait, déplacement ou sélection de colonne, et des modifications des marges des colonnes. Ils écoutent des `ChangeEvent`, des `TableColumnModelEvent` et des `ListSelectionEvent` :

- `void columnAdded(TableColumnModelEvent)`
- `void columnMoved(TableColumnModelEvent)`
- `void columnRemoved(TableColumnModelEvent)`
- `void columnMarginChanged(ChangeEvent)`
- `void columnSelectionChanged(ListSelectionEvent)`

3) Retaillage des colonnes

On peut retailler la table entièrement en retaillant le conteneur dans lequel elle est installée ou en retaillant une (des) colonne(s) individuellement.

Plusieurs stratégies sont possibles :

- globale : en fonction de la propriété `autoResizeMode` (**`int`**, RW, liée) de `JTable`
- locale : en fonction des propriétés `maxWidth`, `minWidth`, `preferredWidth`, `width` (**`int`**, RW, liées) ou `resizable` (**`boolean`**, RW, liée) de `TableColumn`.

3.1) Valeurs de la propriété autoResizeMode

- `AUTO_RESIZE_ALL_COLUMN` : ajustement proportionnel de toutes les colonnes en contrepartie de la colonne en cours d'ajustement
- `AUTO_RESIZE_LAST_COLUMN` : ajustement de la dernière colonne en contrepartie de la colonne en cours d'ajustement
- `AUTO_RESIZE_NEXT_COLUMN` : ajustement de la colonne suivante en contrepartie de la colonne en cours d'ajustement
- `AUTO_RESIZE_OFF` : pas d'ajustement pour compenser la colonne en cours d'ajustement
- `AUTO_RESIZE_SUBSEQUENT_COLUMN` : ajustement proportionnel des colonnes à droite de la colonne en cours d'ajustement

4) Défilement et entête

Une `JTable` doit toujours être associée à un `JScrollPane` car elle ne sait pas gérer le défilement par elle-même et car sa ligne d'entête des colonnes ne peut s'afficher que dans la partie `columnHeader` d'un `JScrollPane`.

Remarque : si l'on ne veut pas d'entête il faut coder

```
table.setTableHeader(null);
table.removeNotify();
```

4.1) Conserver des colonnes fixes

On peut facilement fixer des colonnes à gauche de la `JTable` (comme pour un tableur) en plaçant dans le `rowHeaderView` et dans le corner correspondant du `JScrollPane` une deuxième `JTable`. Les deux `JTable`s doivent alors partager le même modèle de données et le même modèle de sélection.

5) Rendu visuel des cellules

Le rendu visuel des cellules est géré par un composant obtenu d'un `TableCellRenderer` et fournissant une implantation de la méthode :

```
Component getTableCellRendererComponent(
  JTable table,        // la table
  Object value,        // la valeur à afficher
  boolean isSelected, // état de sélection de la cellule
  boolean hasFocus,   // état de focus de la cellule
  int row,            // numéro de ligne de la cellule
  int column          // numéro de colonne de la cellule
)
```

Comme dans le cas des listes, on utilise une seule instance de *renderer* pour dessiner toutes les cellules...

5.1) DefaultCellRenderer

C'est l'implantation de `TableCellRenderer` utilisée pour afficher des `Object`s. Elle dérive `JLabel` et isole la méthode d'affichage dans :

```
protected void setValue(Object v) {
    setText((v == null) ? "" : v.toString());
}
```

Mais une `JTable` n'utilise pas toujours `DTCR` : elle choisit le type de *renderer* en fonction des valeurs retournées par `getColumnClass` (qui est `Object.class` par défaut). On peut donc redéfinir cette méthode (dans le modèle) de sorte qu'elle retourne différentes valeurs, le choix du *renderer* se fera alors ainsi :

getColumnClass	Type de *renderer*	Effet de setValue
`Object.class`	`DefaultTableCellRenderer`	`setText(v.toString())`
`Date.class`	`JTable.DateRenderer`	`setText(DateFormat.format(v))`
`Number.class`	`JTable.NumberRenderer`	`setText(v.toString())`
`Float.class`	`JTable.DoubleRenderer`	`setText(NumberFormat.format(v))`
`Double.class`	`JTable.DoubleRenderer`	`setText(NumberFormat.format(v))`
`Icon.class`	`JTable.IconRenderer`	`setIcon(v)`
`ImageIcon.class`	`JTable.IconRenderer`	`setIcon(v)`
`Boolean.class`	`JTable.BooleanRenderer`	`setSelected(v)`

5.2) Définir ses propres *renderers*

1. Implanter `TableCellRenderer`
2. Associer un *renderer* `r` aux colonnes concernées :

   si une ou plusieurs colonnes sont spécifiques :

   ```
   TableColumnModel tcm = table.getColumnModel();
   TableColumn tc = tcm.getColumn(colIndex);
   // ou aussi tc = table.getColumn(colName);
   tc.setCellRenderer(r);
   ```

   si toutes les colonnes sont du même type `X` :

   Redéfinir `getColumnClass` pour qu'elle retourne `X.class` sur les indices des colonnes de type `X`

   ```
   table.setDefaultRenderer(X.class, r);
   ```

5.3) Algorithme de recherche du *renderer* adéquat

Pour obtenir le *renderer* d'une `TableColumn tc` :

```
Si un renderer a été fixé pour tc[11] Alors
    Utiliser ce renderer
Sinon[12]
    Obtenir le type X des éléments de tc par getColumnClass
    TantQue aucun renderer n'a été fixé pour X Faire
        X <- super type de X
    Fait
    Utiliser ce renderer
FinSi
```

On est assuré de sortir du `TantQue`, au pire sur le type `Object` (par défaut, un `DTCR` est fixé pour `Object`).

6) Editeurs de cellules

Points communs entre éditeur et *renderer* :

- on peut associer un même éditeur à une ou plusieurs colonnes spécifiques (`TableColumn.setCellEditor(TCE)`)
- on peut associer un même éditeur à un type de cellules (`JTable.setDefaultEditor(Class,TCE)`)
- on peut utiliser des composants Swing pour éditer les cellules (`JComboBox`, `JCheckBox`, `JTextField`, ...)

11 par `tc.setCellRenderer(TCR)`
12 par `table.setDefaultRenderer(Class,TCR)`

6.1) Détection de l'action d'édition

Lorsqu'on clique sur une cellule, la table détermine si cette cellule est éditable ou non au moyen de `TableModel.isCellEditable`. Si la réponse est positive, la table retrouve l'éditeur pour cette cellule et lui demande si la cellule doit être éditée ou non (`CellEditor.isCellEditable`). L'éditeur répondra en fonction du nombre de clics et de sa politique d'édition ; l'édition sera possible seulement si les deux réponses sont positives.

6.2) Fonctionnement de l'édition

Lorsque l'utilisateur clique sur une cellule pour l'éditer, si la table accepte l'édition, elle appelle `getTableCellEditorComponent` qui prépare et configure l'éditeur pour refléter correctement le contenu de la cellule à éditer. Puis la table retaille cet éditeur avec les dimensions de la cellule et le superpose au dessus de la cellule pour en capturer l'édition. Enfin, l'éditeur se met en œuvre...

6.3) TableCellEditor

Cette interface étend l'interface `CellEditor` ainsi :

- **`void`** `addCellEditorListener(CellEditorListener)` : enregistre un écouteur auprès de l'éditeur, des notifications auront lieu quand l'éditeur arrêtera ou annulera l'édition
- **`void`** `cancelCellEditing()` : annule l'édition sans prendre en compte la valeur en cours d'édition ; doit notifier les écouteurs enregistrés
- `Object getCellEditorValue()` : la valeur courante stockée par l'éditeur
- **`boolean`** `isCellEditable(EventObject)` : deuxième méthode appelée durant le mécanisme de détection d'édition, l'événement est celui qui est à l'origine de l'édition
- **`void`** `removeCellEditorListener(CellEditorListener)` : supprime l'écouteur de cet éditeur
- **`boolean`** `shouldSelectCell(EventObject)` : indique si la cellule en cours d'édition doit être sélectionnée ou non, l'événement est celui qui est à l'origine de l'édition
- **`boolean`** `stopCellEditing()` : arrête l'édition en prenant en compte la valeur en cours d'édition comme valeur de l'éditeur ; doit notifier les écouteurs enregistrés

Lorsque la table détecte une situation d'édition, elle obtient l'éditeur configuré par appel, sur un `TableCellEditor`, de la méthode :

```
Component getTableCellEditorComponent(
  JTable table,        // la table
  Object value,        // valeur à afficher
  boolean isSelected, // état de sélection de la cellule
  int row,             // ligne de la cellule
  int column           // colonne de la cellule
```

```
)
```

6.4) CellEditorListener

Cet écouteur est doté de deux méthodes réflexes :

- `void editingCanceled(ChangeEvent)` : comportement à adopter quand l'éditeur a annulé l'édition (activée par `CellEditor.cancelCellEditing`)
- `void editingStopped(ChangeEvent)` : comportement à adopter quand l'éditeur a terminé l'édition (activée par `CellEditor.stopCellEditing`)

Normalement, seule la table est un écouteur de son éditeur.

6.4.1) Exemple : un éditeur de dates

```
public class DateEditor extends AbstractCellEditor
    implements TableCellEditor {
  private static final Calendar CAL = Calendar.getInstance();
  private JTextField delegate;
  public DateEditor() {
    delegate = new JTextField();
    delegate.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        DateEditor.this.stopCellEditing();
      }
    });
  }
  public Object getCellEditorValue() {
    CAL.clear();
    try {
      CAL.set(Long.parseLong(delegate.getText()));
    } catch (NumberFormatException e) {
      // rien
    }
    return CAL.getTime();
  }
  public boolean isCellEditable(EventObject e) {
    if (e instanceof MouseEvent) {
      return ((MouseEvent) e).getClickCount() >= 2;
    }
    return false;
  }
  public Component getTableCellEditorComponent( ... ) {
    if (value instanceof Date) {
      delegate.setText(String.valueOf(((Date) value).getTime()));
    } else {
      delegate.setText("0");
    }
    return delegate;
  }
```

```
}
```

6.5) DefaultCellEditor

C'est l'éditeur par défaut pour les cellules de `JTable`s. Il dispose de trois constructeurs :

- `DefaultCellEditor(JCheckBox)` : pour des cellules contenant des booléens, édition sur un seul clic
- `DefaultCellEditor(JComboBox)` : pour des cellules contenant des éléments d'une liste prédéfinie, édition sur un seul clic
- `DefaultCellEditor(JTextField)` : pour des cellules dont la représentation textuelle sera éditée, édition sur double clic.

6.5.1) DefaultCellEditor.EditorDelegate

C'est la classe interne qui gère toutes les commandes d'édition au sein de `DCE`.

- `Object getCellEditorValue()`
- **`void`** `setValue(Object)`
- **`boolean`** `startCellEditing(EventObject)`
- **`void`** `cancelCellEditing()`
- **`boolean`** `stopCellEditing()`
- **`boolean`** `isCellEditable(EventObject)`
- **`void`** `itemStateChanged(ItemEvent)`
- **`boolean`** `shouldSelectCell(EventObject)`

6.5.2) Exemple : édition de date avec un DefaultCellEditor

```
public class DateEditor extends DefaultCellEditor {
  public DateEditor() {
    super(new JTextField());
    final JTextField tf = (JTextField) editorComponent;
    tf.setHorizontalAlignment(JTextField.RIGHT);
    delegate = new EditorDelegate() {
      public Object getCellEditorValue() {
        CAL.clear();
        try {
          CAL.set(Long.parseLong(tf.getText()));
        } catch (NumberFormatException e) {
          // rien
        }
        return CAL.getTime();
      }
      public void setValue(Object value) {
        if (value instanceof Date) {
          tf.setText(String.valueOf(((Date) value).getTime()));
        } else {
          tf.setText("0");
        }
```

```
        }
      };
    }
}
```

7) Sélection

Un certain nombre de propriétés (**`boolean`**, RW, liées) sont disponibles dans `JTable` :

- `rowSelectionAllowed` (défaut : **`true`**) : sélection des lignes
- `columnSelectionAllowed` (défaut : **`false`**) : sélection des colonnes
- `cellSelectionEnabled` (défaut : **`false`**) : sélection des cellules

Si la sélection des lignes et des colonnes est autorisée, alors la sélection des cellules l'est aussi.

7.1) Modèles de sélection

La propriété `selectionModel` de `JTable` (`ListSelectionModel`, RW, liée) contrôle le modèle de sélection des lignes.

La propriété `selectionModel` de `TableColumnModel` (`ListSelectionModel`, RW, non liée) contrôle le modèle de sélection des colonnes.

7.2) Modes de sélection

La propriété `selectionMode` de `JTable` (**`int`**, W, non liée) contrôle à la fois le mode de sélection des lignes et des colonnes.

Pour différencier le comportement ligne/colonne, on utilisera :

```
tbl.setSelectionMode(
    ListSelectionModel.SINGLE_SELECTION
);
tbl.getColumnModel().getSelectionModel().setSelectionMode(
    ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
);
```

8) Ajouter une ligne à une table

Pour ajouter une ligne vierge à une table, après que la table aura été affichée, il faut un modèle mutable (comme `DefaultTableModel`) doté d'une méthode `addRow` par exemple.

Ensuite, pour s'assurer que la ligne deviendra visible juste après l'ajout, il faut rajouter un écouteur de modification du modèle de données qui va faire défiler les lignes jusqu'au bon endroit. On utilisera la méthode `scrollRectToVisible` de la classe `JComponent` (*cf.* JavaDoc et TP).

Chapitre 6 : composant JTree

Il s'agit d'un composant graphique définissant une vue sur un modèle arborescent de données.

1) Généralités

L'accès à un nœud se fait soit par son chemin à partir du nœud racine (une instance de `TreePath`), soit par un numéro de ligne d'affichage (attention, le numéro de ligne d'un nœud visible change en fonction du pliage des différents nœuds de l'arbre).

Les nœuds internes peuvent se trouver dans l'un ou l'autre des deux états suivants :

- déplié (*expanded*) : nœud interne dont les enfants directs sont exposés à la vue (mais pas nécessairement dans la zone visible de l'arbre)
- plié (*collapsed*) : nœud interne non déplié

Les nœuds affichés à l'écran peuvent être qualifiés de

- affichable (*viewable*) : nœud dont tous les ancêtres sont dépliés
- masqué (*hidden*) : nœud dont un ancêtre au moins est plié
- visible (*displayed*) : nœud affichable et dans la zone visible de l'arbre

JTree contient quelques méthodes utilitaires en rapport avec le pliage/dépliage des nœuds :

- `boolean isCollapsed(TreePath)`
- `boolean isExpanded(TreePath)`
- `void collapsePath(TreePath)`
- `void expandPath(TreePath)`

La méthode `void makeVisible(TreePath)` rend affichable le nœud de chemin donné et `boolean isVisible(TreePath)` indique si le nœud de chemin donné est affichable.

Voici quelques exemples de mises en place de JTree très simples :

- avec des maps

```
Hashtable m = new Hashtable();
Hashtable m2 = new Hashtable();
m.put("Yoda", new Hashtable());
m.put("Yan", new Hashtable());
m.put("Dark", m2); {
    m2.put("Luke", new Hashtable());
    m2.put("Léia", new Hashtable());
}
m.put("D2R2", new Hashtable());
m.put("C6PO", new Hashtable());
m.put("Chewee", new Hashtable());
JTree tree = new JTree(m);
```

- avec un tableau d'objets

```
Object[] m = new Object[] {
    "Yoda", "Yan",
    new Object[] {"Luke", "Léia" },
    "D2R2", "C6PO", "Chewee"
};
JTree tree = new JTree(m);
```

- avec des vecteurs

```
Vector m = new Vector();
Vector m2 = new Vector() {
    public String toString() {
        return "Dark";
    }
};
m.add("Yoda");
m.add("Yan");
m.add(m2); {
    m2.add("Luke");
    m2.add("Léia");
}
m.add("D2R2");
m.add("C6PO");
m.add("Chewee");
JTree tree = new JTree(m);
```

2) Gestion du défilement

Comme pour tous les composants qui implantent `Scrollable`, un `JTree` a besoin d'un `JScrollPane` pour gérer son défilement. De plus :

- `void scrollPathToVisible(TreePath)` : assure qu'un nœud de chemin donné est visible
- `void scrollRowToVisible(int)` : assure qu'une ligne donnée (à partir de 0) est visible
- `boolean isRootVisible()` : indique si la racine est visible
- `void setRootVisible(boolean)` : rend la racine visible ou non
- `int getVisibleRowCount()` : indique le nombre de lignes visibles
- `void setVisibleRowCount(int)` : fixe le nombre de lignes visibles

Rappel : une ligne n'affiche pas toujours le même nœud, tout dépend du pliage des nœuds intermédiaires !

3) Modèle de données

Le modèle des données d'un `JTree` est défini par `TreeModel` :

- `Object getChild(Object parent, int index)` : le fils à la position `index` du nœud `parent`, `index` représente la position du fils par rapport à `parent` lors de l'affichage (à partir de 0)
- `int getChildCount(Object parent)` : le nombre de fils du nœud `parent`
- `int getIndexOfChild(Object parent, Object child)` : la position du nœud `child` dans le nœud `parent` (toujours par rapport à l'affichage), -1 s'il n'y a pas de relation de filiation
- `Object getRoot()` : le nœud racine de l'arbre
- `boolean isLeaf(Object node)` : indique si le nœud `node` est une feuille
- `void valueForPathChanged(TreePath p, Object v)` : doit être exécutée pour signaler que l'utilisateur a modifié la valeur du nœud identifié par `p`, si `v` est réellement une nouvelle valeur, le modèle devrait notifier les écouteurs de modification à l'aide de `fireTreeNodesChanged`
- `void addTreeModelListener(TreeModelListener)` : ajout d'un écouteur de modification de ce modèle
- `void removeTreeModelListener(TreeModelListener)` : suppression d'un écouteur de modification de ce modèle

4) Mise en place

4.1) avec un modèle

On peut fournir le modèle au constructeur (`JTree(TreeModel)`), le composant ainsi construit affiche sa racine, contrairement aux mises en places simples exposées plus haut.

4.2) avec une racine

On peut aussi laisser le constructeur créer son propre modèle (de type `DefaultTreeModel`) à partir de la racine passée en argument (`JTree(TreeNode, boolean)`), le paramètre booléen définit la politique de détermination des feuilles :

- `false` : seuls les nœuds qui n'ont pas de fils sont des feuilles (`getChildCount() == 0`)
- `true` : seuls les nœuds qui ne peuvent pas avoir de fils sont des feuilles (`getAllowsChildren() == false`)

La forme simplifiée `JTree(TreeNode)` équivaut à `JTree(TreeNode, false)`.

Le composant ainsi construit affiche sa racine, contrairement aux mises en places simples exposées plus haut.

5) Gestion des noeuds

Deux interfaces : une pour les nœuds non mutables, une autre pour les nœuds mutables ;

une seule classe d'implantation de nœuds mutables.

5.1) interface TreeNode

L'interface `TreeNode` définit l'ensemble des services que propose un nœud au modèle qui gère ce nœud (et tous ceux de sa famille). Ces nœuds sont non mutables, au sens où ils ne peuvent pas changer de valeur ni de position dans l'arbre.

- `Enumeration children()` : l'énumération des fils de ce nœud
- `boolean getAllowsChildren()` : indique si ce nœud peut porter des fils
- `TreeNode getParent()` : le nœud père de ce nœud
- `TreeNode getChildAt(int childIndex)` : le nœud fils d'index `childIndex`, utilisée par `TreeModel.getChild`
- `int getChildCount()` : le nombre de fils de ce nœud, utilisée par `TreeModel.getChildCount`
- `int getIndex(TreeNode node)` : l'index de `node` pour ce nœud, utilisée par `TreeModel.getIndexOfChild`
- `boolean isLeaf()` : indique si ce nœud est une feuille, utilisée par `TreeModel.isLeaf`

5.2) interface MutableTreeNode

L'interface `MutableTreeNode` étend l'interface `TreeNode` pour prendre en compte la variabilité des nœuds en termes de valeur et de position dans l'arbre. Elle est implémentée par la classe `DefaultMutableTreeNode`.

- `void insert(MutableTreeNode child, int index)` : insère `child` en position `index` comme fils de ce nœud, devrait détacher `child` de son éventuel ancien père et lui attacher `this` comme nouveau père
- `void remove(int index)` : supprime le fils en position `index` pour ce nœud, devrait détacher le fils en position `index` de ce nœud
- `void remove(MutableTreeNode node)` : supprime `node` des fils de ce nœud, devrait détacher `node` de ce nœud
- `void removeFromParent()` : détache ce nœud de son père
- `void setParent(MutableTreeNode newParent)` : attache ce nœud à `newParent`
- `void setUserObject(Object object)` : fixe la nouvelle étiquette portée par ce nœud à `object`

5.3) classe TreePath

Représente la notion de chemin à partir de la racine dans un arbre. C'est la seule méthode sûre pour identifier un nœud car, selon l'état de pliage des nœuds de l'arbre, l'index d'un

nœud peut être indéterminé ou peut changer au cours du temps...

L'implémentation est réalisée sous forme d'un tableau de nœuds, ordonné depuis la racine de l'arbre jusqu'au dernier nœud du chemin. Dans la documentation, on parle de chemin descendant avec la signification suivante :

> p2 descend de p1 <==> p1 est préfixe de p2.

Les méthodes de cette classes :

- `Object getLastPathComponent()` : le dernier nœud de ce chemin
- `TreePath getParentPath()` : un nouveau chemin constitué des nœuds de ce chemin sauf le dernier
- `Object[] getPath()` : ce chemin sous forme de tableau ordonné
- `Object getPathComponent(int index)` : le nœud de ce chemin situé en position `index`
- `int getPathCount()` : le nombre de nœuds de ce chemin
- `boolean isDescendant(TreePath tp)` : indique si `tp` est un descendant de ce chemin
- `TreePath pathByAddingChild(Object child)` : un nouveau chemin constitué des nœuds de ce chemin plus `child` à la fin

6) Gestion des événements de modification du modèle de données

On dispose de l'interface `TreeModelListener` pour écouter les modifications, insertions et suppressions de nœuds dans un modèle. Attention : les modifications de la structure lorsqu'elle sont effectuées par le modèle sont bien automatiquement notifiées aux écouteurs mais pas lorsqu'elles sont effectuées par l'intermédiaire des nœuds !

Ce mode de fonctionnement est justifié par le fait qu'il permet d'effectuer plusieurs modifications suivies d'une seule notification (laissée à l'appréciation du programmeur, pour plus d'efficacité). Son inconvénient majeur : le programmeur ne doit pas oublier les notifications !

6.1) Exemples de code

Différence entre modification du modèle et modification des nœuds :

```
DefaultTreeModel model = (DefaultTreeModel) tree.getModel();
for (int i = 0; i < nodesToInsert.length; i++) {
    model.insertNodeInto(nodesToInsert[i], parent, i);
}
-------------------------------------------------------------
DefaultTreeModel model = (DefaultTreeModel) tree.getModel();
int[] indices = new int[nodesToInsert.length];
for (int i = 0; i < nodesToInsert.length; i++) {
```

```
    parent.insert(nodesToInsert[i], i);
    indices[i] = i;
}
model.nodesWhereInserted(parent, indices);
```

6.2) interface TreeModelListener

- `void treeNodesChanged(TreeModelEvent evt)` : changement d'état des fils de `evt.getTreePath()` donnés par `evt.getChildren()`, la structure de l'arbre n'a pas changée
- `void treeStructureChanged(TreeModelEvent evt)` : modification importante de la structure de l'arbre sous le nœud `evt.getPath()`
- `void treeNodesInserted(TreeModelEvent evt)` : insertion des nœuds `evt.getChildren()` sous le nœud `evt.getPath()`
- `void treeNodesRemoved(TreeModelEvent evt)` : suppression des nœuds `evt.getChildren()` sous le nœud `evt.getPath()`

6.3) classe TreeModelEvent

- `Object[] getPath()` : chemin d'accès au nœud sous lequel a eu lieu la modification, donné sous forme de tableau
- `TreePath getTreePath()` : chemin d'accès au nœud sous lequel a eu lieu la modification, donné sous forme de `TreePath`
  - pour `treeStructureChanged`, retourne l'ancêtre commun à tous les nœuds modifiés
  - pour `treeNodesChanged`, retourne la racine de l'arbre
- `int[] getChildIndices()` : tableau des positions pour lesquelles les enfants ont été modifiés
  - supression : indique les indices auxquels les enfants ont été enlevés par rapport au tableau initial
  - insertion : indique les indices auxquels les enfants ont été ajoutés par rapport au tableau final
  - pour `treeStructureChanged`, retourne les indices où il y a eu modification
  - pour `treeNodesChanged`, retourne `null`
- `Object[] getChildren()` : tableau des enfants du nœud identifié par `getPath()` aux positions indiquées par `getChildIndices()`

7) Rendu graphique des noeuds

Le rendu graphique se fait par l'intermédiaire de l'interface `TreeCellRenderer` qui ne contient que la méthode :

```
Component getTreeCellRendererComponent(
    JTree tree,        // arbre
    Object value,      // le nœud à afficher
    boolean selected,  // le nœud est-il sélectionné ?
    boolean expanded,  // le nœud est-il déplié ?
    boolean leaf,      // est-ce une feuille ?
    int row,           // ligne d'affichage
    boolean hasFocus   // a-t-il le focus ?
)
```

La classe `DefaultTreeCellRenderer` dérive `JLabel` et implémente cette interface.

7.1) Calcul du rendu d'un noeud

Tout d'abord, un nœud doit savoir afficher son étiquette avec `toString`. Ensuite, le calcul du texte à afficher se fait ainsi :

```
Le TreeUI appelle getTreeCellRendererComponent sur le renderer
    le renderer appelle convertValueToText(value) sur le JTree
        le JTree appelle toString() sur value
    le renderer modifie son texte avec la valeur retournée
```

Pour modifier ce comportement par défaut, il faut dériver `JTree` et redéfinir la méthode `convertValueToText`.

7.2) Changer l'affichage des noeuds sans changer de renderer

On peut apporter certaines modifications à la configuration d'affichage en utilisant les propriétés suivantes de `DTCR` :

- `backgroundNonSelectionColor` (`Color`, RW)
- `backgroundSelectionColor` (`Color`, RW)
- `borderSelectionColor` (`Color`, RW)
- `closedIcon` (`Icon`, RW)
- `defaultClosedIcon` (`Icon`, R)
- `defaultLeafIcon` (`Icon`, R)
- `defaultOpenIcon` (`Icon`, R)
- `font` (`Font`, RW)
- `leafIcon` (`Icon`, RW)
- `openIcon` (`Icon`, RW)
- `textNonSelectionColor` (`Color`, RW)
- `textSelectionColor` (`Color`, RW)

7.3) Changer de renderer

Sur un exemple :

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer {
  private Map<String, ImageIcon> icons;
  public MyTreeCellRenderer(Map<String, ImageIcon> icons) {
```

```
    this.icons = icons;
    setBackgroundSelectionColor(Color.BLACK);
    setTextSelectionColor(Color.WHITE);
    setFont(new Font("Papyrus", ...));
  }
  public Component getTreeCellRendererComponent(
      JTree tree, Object value, boolean sel,
      boolean expanded, boolean leaf, int row,
      boolean hasFocus) {
    ImageIcon i = (ImageIcon) icons.get(value.toString());
    if (leaf) {
      setLeafIcon(i);
    } else if (expanded)
      setOpenIcon(i);
    } else {
      setClosedIcon(i);
    }
    return super.getTreeCellRendererComponent(
        tree, value, sel, expanded, leaf, row, hasFocus);
  }
}
-----------------------------------------------------------
Map<String, ImageIcon> icons = ...;
JTree tree = new JTree(new Object {...});
tree.setCellRenderer(new MyTreeCellRenderer(icons));
```

7.4) Modification des poignées

Les icônes des poignées ne sont pas gérées par le renderer mais par le *UI-delegate* (instance de `BasicTreeUI`). Pour supprimer les poignées il faut déterminer la classe exacte du *UI-delegate* (`MetalTreeUI`, `MotifTreeUI`, ou `WindowsTreeUI`) puis dériver cette classe en y redéfinissant la méthode **`boolean`** `shouldPaintExpandControlMethod()` pour qu'elle retourne **`false`**.

7.4.1) Modification des poignées d'un JTree particulier

- `((BasicTreeUI) tree.getUI()).setExpandedIcon(Icon)`
- `((BasicTreeUI) tree.getUI()).setCollapsedIcon(Icon)`
- `((BasicTreeUI) tree.getUI()).setLeftChildIndent(int)`
- `((BasicTreeUI) tree.getUI()).setRightChildIndent(int)`

7.4.2) Modification des poignées de tous les JTree

On peut utiliser les propriétés de L&F des `BasicTreeUI` gérées par le `UIManager` :

- `UIManager.put("Tree.expandedIcon", Icon)`
- `UIManager.put("Tree.collapsedIcon", Icon)`

- `UIManager.put("Tree.leftChildIndent", Integer)`
- `UIManager.put("Tree.rightChildIndent", Integer)`

8) Edition des noeuds

Un éditeur de nœuds est une instance de `TreeCellEditor`. La classe `DefaultTreeCellEditor` fournit une implémentation par défaut de cette interface. Elle utilise un `TreeCellRenderer` pour afficher l'icône du nœud et un délégué de type `DefaultCellEditor` pour éditer le texte du nœud.

Le `DefaultTreeCellEditor` se construit avec :

- un `JTree`
- un `DefaultTreeCellRenderer`
- [un `TreeCellEditor`]

L'instance créée affichera toujours l'icône du *renderer* et utilisera le `JTextField` du `DefaultCellEditor` pour éditer l'étiquette du nœud.

Le contrôle de l'édition des noeuds se fait au niveau du `JTree` :

- Contrôle global : la propriété `editable` (**`boolean`**, RW, liée) gère globalement l'éditabilité de tous les nœuds.
- Contrôle au cas par cas : la méthode **`boolean`** `isPathEditable(TreePath)` est appelée au début du calcul de l'éditabilité d'un nœud. Elle retourne la valeur de `isEditable()` par défaut mais est adaptable par redéfinition en installant un algorithme filtrant les nœuds éditables de ceux qui ne le sont pas.

9) Sélection des noeuds

La sélection des nœuds se fait à l'aide d'un `DefaultTreeSelectionModel`. Il y a une propriété associée au JTree :

- `selectionModel` (`TreeSelectionModel`, RW, liée)

et une autre associée au modèle de sélection :

- `selectionMode` (**`int`**, RW, non liée)

On peut supprimer la possibilité de sélectionner les nœuds d'un arbre en mettant sa propriété `selectionModel` à **`null`**.

9.1) Modes de sélection des noeuds

Les différentes valeurs possibles sont les constantes suivantes de `TreeSelectionModel` :

- `CONTIGUOUS_TREE_SELECTION` : une seule zone contiguë de nœuds sélectionnés

- `DISCONTIGUOUS_TREE_SELECTION` : plusieurs zones de nœuds sélectionnés
- `SINGLE_TREE_SELECTION` : un seul nœud sélectionné

Sélectionner un nœud interne ne sélectionne pas ses descendants. Par ailleurs, la sélection d'une zone contiguë sur plusieurs niveaux n'en sélectionne que les nœuds visibles.

9.2) Notification de sélection

On écoute les événements de sélection avec des `TreeSelectionListener` dotés d'une seule méthode réflexe `void valueChanged(TreeSelectionEvent)`. Ces écouteurs peuvent être attachés au `JTree` directement, auquel cas la source de l'événement est un `JTree`, ou bien à son modèle de sélection, auquel cas la source de l'événement est un `TreeSelectionModel`.

Le comportement des `TreeSelectionEvent` est défini par :

- `Object cloneWithSource(Object newSource)` : une copie de l'événement dont la source est `newSource`
- `TreePath getNewLeadSelectionPath()` : le chemin le plus bas dans la sélection après le changement de sélection
- `TreePath getOldLeadSelectionPath()` : le chemin le plus bas dans la sélection avant le changement de sélection
- `TreePath getPath()` : le chemin ajouté à ou supprimé de la sélection
- `TreePath[] getPaths()` : les chemins ajoutés à ou supprimés de la sélection
- `boolean isAddedPath()` : indique si le chemin a été ajouté à (`true`) ou supprimé de (`false`) la sélection
- `boolean isAddedPath(int index)` : indique si le chemin en position `index` dans la sélection a été ajouté (`true`) ou supprimé (`false`)
- `boolean isAddedPath(TreePath path)` : indique si `path` a été ajouté à (`true`) ou supprimé de (`false`) la sélection

9.3) Sélection par programmation

On peut sélectionner les nœuds en intervenant directement sur l'arbre qui les porte, en modifiant les propriétés de `JTree` suivantes :

- `lastSelectedPathComponent` (`Object`, R, non liée)
- `anchorSelectionPath` (`TreePath`, RW, liée)
- `leadSelectionPath` (`TreePath`, RW, liée)
- `selectionCount` (`int`, R, non liée)
- `selectionEmpty` (`boolean`, R, non liée)
- `selectionPath` (`TreePath`, RW, non liée)
- `selectionPaths` (`TreePath[]`, RW, non liée)

- `leadSelectionRow` (**`int`**, R, non liée)
- `maxSelectionRow` (**`int`**, R, non liée)
- `minSelectionRow` (**`int`**, R, non liée)
- `selectionRow` (**`int`**, W, non liée)
- `selectionRows` (**`int`**`[]`, RW, non liée)

10) Gestion du pliage

Lorsqu'on veut programmer le (dé)pliage des nœuds internes, on utilise les méthodes suivantes de `JTree` :

- **`void`** `collapsePath(...)`
- **`void`** `collapseRow(...)`
- **`void`** `expandPath(...)`
- **`void`** `expandRow(...)`

Lorsqu'on veut détecter les (dé)pliages, on peut placer les détections juste avant (`TreeWillExpandListener`) ou juste après (`TreeExpansionListener`).

10.1) interface TreeWillExpandListener

- **`void`** `treeWillCollapse(TreeExpansionEvent)` **`throws`** `ExpandVetoException`
- **`void`** `treeWillExpand(TreeExpansionEvent)` **`throws`** `ExpandVetoException`

L'intérêt des `TreeWillExpandListener` tient à ce qu'ils permettent d'accepter/refuser le (dé)pliage, les exceptions servant à arrêter l'action. Pour tester si l'action a aboutie, on utilisera les méthodes de `JTree` :

- **`boolean`** `isExpanded(int)`
- **`boolean`** `isExpanded(TreePath)`
- **`boolean`** `isCollapsed(int)`
- **`boolean`** `isCollapsed(TreePath)`

C'est ce que l'on mettra en place, par exemple, lorsqu'on veut peupler un nœud interne juste avant le moment où le client va le déplier...

10.2) interface TreeExpansionListener

- **`void`** `treeCollapsed(TreeExpansionEvent)`
- **`void`** `treeExpanded(TreeExpansionEvent)`

Architecture des applications graphiques en Java

Philippe Andary

janvier 2023

1 Introduction

Quand on construit du logiciel objet qui présente ses données à travers une interface graphique, il est possible d'organiser les classes qui le constituent en trois catégories, chacune dédiée à une fonctionnalité bien précise : modèle, vue ou contrôleur (MVC). Ceci dans le but d'optimiser les qualités de lisibilité, de simplicité, d'extensibilité et de maintenance du logiciel.

Dès que le moteur applicatif est un peu conséquent, mais surtout d'un fonctionnement lent par rapport au rendu graphique de l'application, on utilise la notion de fil d'exécution (thread) afin d'améliorer la fluidité du fonctionnement de l'application graphique. Or, en Java, la bibliothèque Swing de composants graphiques n'est pas sûre relativement aux accès concurrents à ses éléments. Plus précisément, il ne faut accéder aux composants Swing qu'à travers un seul thread : le thread de gestion des événements (*Event Dispatch Thread*, noté EDT dans la suite) qui s'occupe aussi du dessin des composants graphiques de l'application.

Ce document présente une architecture possible, en Java, pour de telles applications graphiques : le moteur applicatif utilise les threads autant que de besoin et de manière disciplinée, afin de ne pas figer l'affichage de l'interface graphique, tout en assurant que la manipulation des composants graphiques et de leurs modèles est bien réalisée sur EDT. Comme on le verra, cette architecture ne nécessite pas la délicate mise en place de synchronisation : elle se suffit du mécanisme de gestion des événements propre à Java.

La section 2 rappelle comment on organise une application graphique en respectant l'architecture MVC. La section 3 aborde l'architecture à modèle séparable des composants Swing, en remarquant qu'elle n'est qu'une version ad'hoc de la précédente. Enfin, la section 4, après avoir rappelé que la bibliothèque Swing n'est pas sûre dans le cadre de la programmation concurrente et avoir posé les bases du problème qui nous intéresse, expose une manière sûre de gérer l'activité du modèle de l'application sur différents threads, en fonction de la complexité structurelle de celui-ci.

2 Architecture MVC

2.1 MVC à l'origine

Au sein d'une application développée selon une architecture MVC, le code est organisé en trois parties distinctes qui collaborent étroitement ([5], [1]). Elles concernent respectivement la gestion des données (modèle), le rendu graphique (vue) et la communication entre l'application et l'utilisateur (contrôleur).

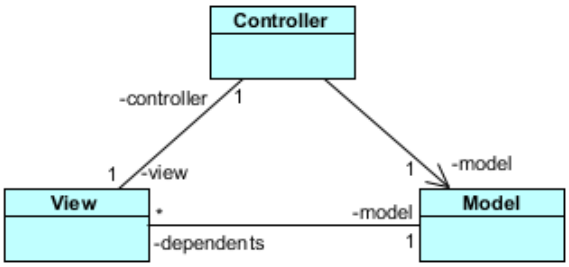

FIGURE 1 – Relations entre les classes de l'architecture MVC d'origine.

— Le *modèle* gère la logique applicative du logiciel, c'est-à-dire qu'il encapsule les données nécessaires à l'application, le moyen d'y accéder et la manière de les modifier. Lorsque ses données changent de valeur, le modèle notifie les vues préalablement enregistrées afin qu'elles puissent se mettre à jour.
— La *vue* gère le rendu graphique de l'application, c'est-à-dire qu'elle spécifie la manière dont il faut afficher les données que contient le modèle, auxquelles elle accède à travers les services que propose ce dernier. Elle se déclare auprès du modèle qui pourra ainsi la notifier lorsqu'une mise à jour sera nécessaire. Enfin, elle transmet les actions de l'utilisateur au contrôleur.
— Le *contrôleur* définit le comportement de l'application. Il reçoit les actions de l'utilisateur qui lui sont transmises par la vue, récupère éventuellement des informations à partir de la vue et traduit ces actions ainsi paramétrées en modifications adaptées pour le modèle. Il peut arriver aussi qu'il commande directement la vue sans passer par le modèle dans certains cas simples.

L'association entre une vue et un contrôleur est unique et bidirectionnelle, ils se connaissent donc mutuellement.

À un modèle on peut associer plusieurs vues qui le représenteront, chacune totalement ou en partie seulement. Le mode de notification des vues par le modèle peut être de deux sortes :

— *push*, la notification contient toutes les informations nécessaires à la mise à jour de la vue ;
— *pull*, la vue devra consulter le modèle pour y puiser les informations nécessaires à son rafraîchissement.

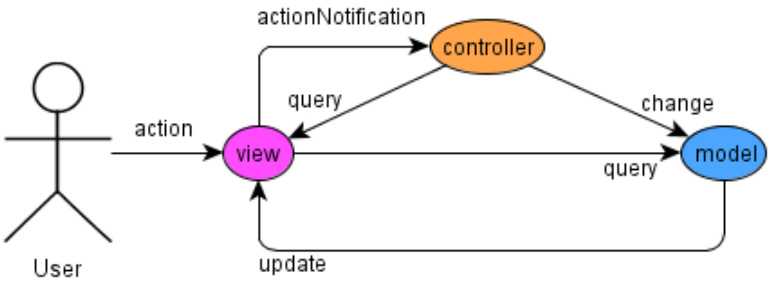

FIGURE 2 – Collaboration entre les éléments de l'architecture MVC d'origine.

Il faut remarquer que l'implantation, dans le modèle, du mécanisme de notification des vues est nécessaire dès lors que d'autres objets que les contrôleurs peuvent changer l'état de celui-ci puisque c'est alors le seul moyen efficace d'assurer la cohérence entre l'état du modèle et l'apparence de la vue. Dans le cas extrême où le contrôleur serait le seul à modifier le modèle, la relation entre ce dernier et la vue qui en dépend pourrait disparaître sous réserve que le contrôleur rafraîchisse la vue à chaque fois que le modèle est modifié. En dehors du mécanisme de notification, le modèle n'a pas connaissance du comportement de ses vues ni de celui des contrôleurs qui leurs correspondent.

Cette description correspond à l'architecture d'origine, telle qu'elle était réalisée en Smalltalk-80. Dans ce langage, MVC était implanté directement dans des classes bas-niveau (`View`, `Controller`, `Model` et `Object`) qu'il suffisait de réutiliser par héritage.

2.2 MVC en Java

Dans un langage comme Java, les développeurs sont fortement incités à réutiliser ce type d'architecture ([2]) en utilisant le patron de conception OBSERVER ([3]). Le mécanisme de notification (de type *pull*) est alors encapsulé dans une classe `Observable` dont devra dériver le modèle. La dépendance entre le modèle et ses vues est réduite à sa plus simple expression par utilisation de l'interface `Observer`. Ces deux types sont présents dans le paquetage `java.util`. Concernant la relation vue-contrôleur, c'est la notion d'écouteur d'événements qui est utilisée. Un écouteur est un objet très léger (souvent instance d'une classe locale anonyme dont le code est très court) dont le comportement est réduit à la gestion d'un type précis d'actions de l'utilisateur comme la manipulation de la souris, ou celle du clavier, mais pas les deux ensemble. La notion de contrôleur se trouve donc éclatée en deux parties : un observateur (pour la relation contrôleur-vue utilisant le modèle) et un écouteur (pour la relation contrôleur-modèle utilisant la vue). Les relations entre les différentes classes et interfaces sont résumées dans la figure 3.

Par ailleurs, les notifications d'événements de bas-niveau (clics souris, frappe de touche au clavier, gestion des fenêtres) se font de manière événementielle, c'est-à-dire pilotées par un thread dédié (EDT) qui puise indéfiniment dans une file d'objets (les événements), elle-même alimentée par le noyau système, le système de fenêtrage ou l'application. Le

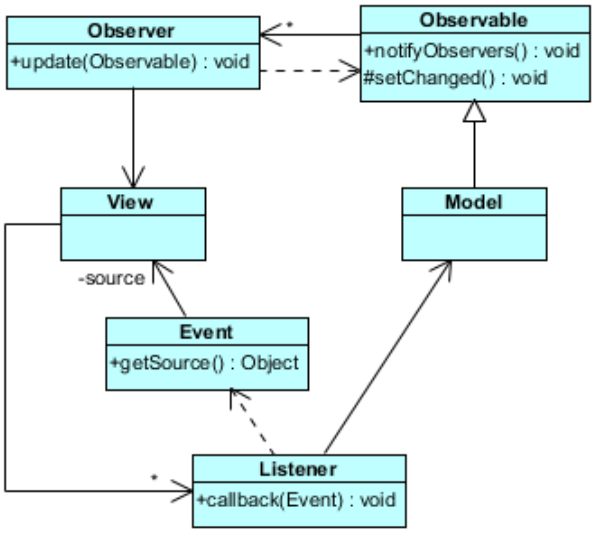

FIGURE 3 – Relations entre les classes de l'architecture MVC en Java.

diagramme de collaboration entre les différents objets est présenté à la figure 4. On obtient ainsi une version adaptée à Java de l'architecture MVC.

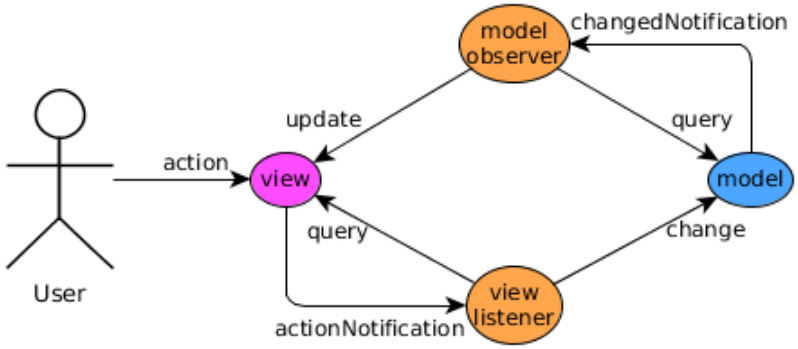

FIGURE 4 – Collaboration entre les éléments de l'architecture MVC en Java.

Cette fois, on notera bien que la vue et le modèle sont complètement indépendants l'un de l'autre ; il suffit que la vue puisse gérer une séquence d'écouteurs (mécanisme implanté dans les composants graphiques de l'API) et le modèle une séquence d'observateurs (mécanisme implanté dans la classe `Observable` de l'API). Enfin, puisque la vue agit sur l'écouteur associé à travers le mécanisme de gestion des événements, l'ensemble des actions de ce schéma se déroule par défaut sur EDT.

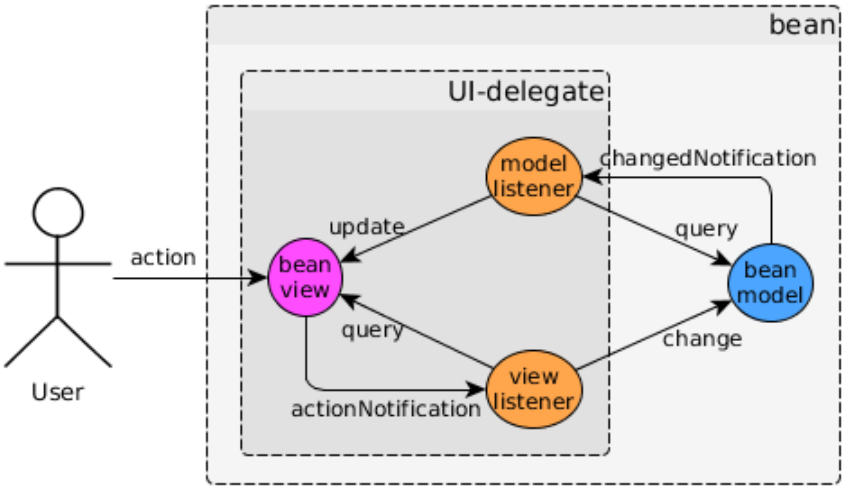

FIGURE 5 – Architecture à modèle séparable des composants Swing.

3 Swing et MVC

Dans ses différents composants graphiques, Swing regroupe par le biais de classes internes les parties vue et contrôleur en une entité unique : le gestionnaire graphique (*UI-delegate*). Cette architecture dérivée de MVC (dite « à modèle séparable ») permet de minimiser l'interface entre vue et contrôleur, tout en mettant la puissance de MVC au service du développement du composant (voir figure 5).

Lorsqu'on utilise la bibliothèque de composants Swing en Java, on ne fait plus de différence entre écouteur et observateur : tout n'est qu'écouteur. Un mécanisme de notification de type *pull* peut être réalisé au sein du modèle via une implantation idoine de l'interface `ChangeListener` et l'utilisation de la classe `EventListenerList`. Cette implantation est légère car elle permet de n'instancier qu'un seul événement par modèle, à réutiliser lors de chaque notification. Elle est à privilégier dans le cas où les notifications par le modèle sont fréquentes.

De plus, un mécanisme de notification de type *push* peut être facilement réalisé par la mise en place de propriétés (au sens de la norme JavaBean) observables à l'aide d'instances de `PropertyChangeListener`, le mécanisme de notification étant lui-même encapsulé dans la classe `PropertyChangeSupport`.

Il est ici important de remarquer que les interactions entre les différents protagonistes sont toutes initiées par un événement bas-niveau émis par la vue vers ses écouteurs suite à l'action de l'utilisateur. Par conséquent les accès à la vue et à son modèle se font uniquement sur EDT.

Si, dans ces conditions, un thread autre que EDT était à l'origine d'une modification du modèle, la vue serait mise à jour sur cet autre thread. Comme on va le voir dans la section suivante, cela n'est pas permis. C'est d'ailleurs à cause de cette interdiction, et pour être certain que tous les composants graphiques sont bien réalisés sur EDT (et pas sur le *main*

thread), que toutes nos applications se lancent à l'aide de la « formule magique » :

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        ... code de création et de démarrage de l'application
    }
}
```

4 Swing et la programmation concurrente

Swing n'est pas *thread-safe*[1]. Cela signifie que l'invocation de méthodes sur les composants graphiques ou sur leurs modèles, à partir de plusieurs threads distincts, risque de provoquer des problèmes d'interférence entre les threads et des erreurs liées à l'incohérence des mises à jour de la mémoire. Pratiquement, cela a pour conséquence l'obligation d'accéder aux composants graphiques (ainsi qu'à leurs modèles) à l'aide d'un unique thread : EDT.

Le problème ? Une application graphique dont la logique métier est un peu lourde va avoir tendance à geler l'interface graphique. Par exemple, télécharger un gros fichier sur le Web prend plusieurs minutes ; or si le téléchargement est initié par un clic souris sur un bouton, c'est EDT qui va être spontanément sollicité et pendant ce temps-là, l'interface graphique ne répondra plus puisque EDT est accaparé par une tâche longue...

Par conséquent, il est normal et souhaitable qu'une tâche lourde ne soit pas exécutée sur EDT mais sur un thread utilisateur (que nous appellerons UT, pour *user thread*). Par contre, si au cours de l'exécution de cette tâche il est nécessaire de manipuler des composants Swing ou leurs modèles, il faudra revenir le faire sur EDT.

4.1 Architecture supportant la concurrence (cas simple)

D'une manière générale, l'envoi d'un message asynchrone en Java se fait par le truchement d'un thread dont le code cible contient l'envoi du message. Lorsque ce thread est EDT, ce code cible est encapsulé dans une instance d'`ActiveEvent` (à ne pas confondre avec `ActionEvent`) qui est placée sur la file des événements (de type `EventQueue`) que gère EDT. Dit autrement, on fait appel à la méthode `SwingUtilities.invokeLater`), le code correspondant au message devant être codé dans la méthode `run` d'une instance de `Runnable`.

Dans les cas où le modèle doit traiter de lourdes tâches, on aboutit donc au diagramme de la figure 6. La flèche en pointillés étiquetée par *change* indique un envoi de message asynchrone. L'étiquette rouge sur cette flèche précise la qualité du thread utilisé pour rendre le message asynchrone.

Dans les cas les plus simples on utilisera le schéma de code suivant :

1. On pourrait se demander pourquoi ? Un élément de réponse dans [4].

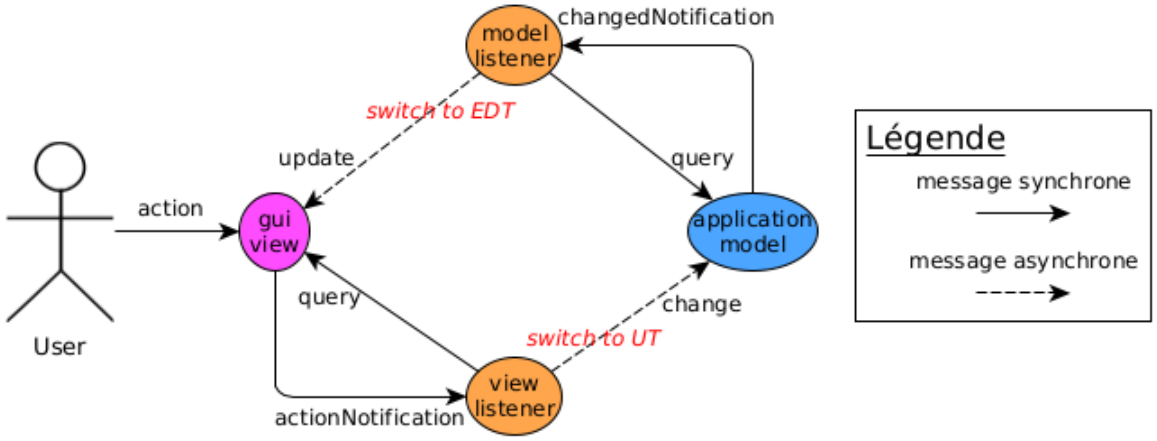

FIGURE 6 – Collaboration entre les éléments de l'architecture MVC supportant la concurrence (modèle simple).

```
// dans view listener
final ViewInfo vi = guiView.query(...);
Runnable target = new Runnable() {
    public void run() {
        applicationModel.change(vi);
    }
};
new Thread(target).start(); // exécution de target.run sur UT
---------------><8
// dans model listener
final ModelInfo mi = applicationModel.query(...);
Runnable target = new Runnable() {
    public void run() {
        guiView.update(mi);
    }
};
SwingUtilities.invokeLater(target); // exécution de target.run sur EDT
```

Alternativement, Swing propose d'utiliser une classe abstraite générique : `SwingWorker`. Une instance de cette classe permet d'exécuter ailleurs que sur EDT la tâche codée dans la méthode `SwingWorker.doInBackground` (qui joue le rôle de la méthode `run` du code cible d'un thread, mais qui permet de plus le calcul d'une valeur qu'elle retournera). Cette méthode peut appeler `SwingWorker.publish` pour communiquer des données qui seront traitées sur EDT par appel à `SwingWorker.process`. Il suffit alors de définir cette dernière de sorte qu'elle implante le comportement souhaité pour la mise à jour des composants graphiques. La tâche se termine par appel de `SwingWorker.done` (qui ne fait rien par

défaut, mais qui peut-être redéfinie) et le fruit du calcul de `doInBackground` est accessible par `SwingWorker.get`, qui est une méthode bloquante (donc attention si cette dernière méthode est appelée sur EDT).

4.2 Architecture supportant la concurrence (cas complexe)

Que l'on utilise `SwingWorker` ou que l'on gère soi-même ses propres threads, il peut arriver que le modèle de l'application soit suffisamment complexe et consommateur de temps CPU pour que l'on soit amené à découper l'application comme sur la figure 7.

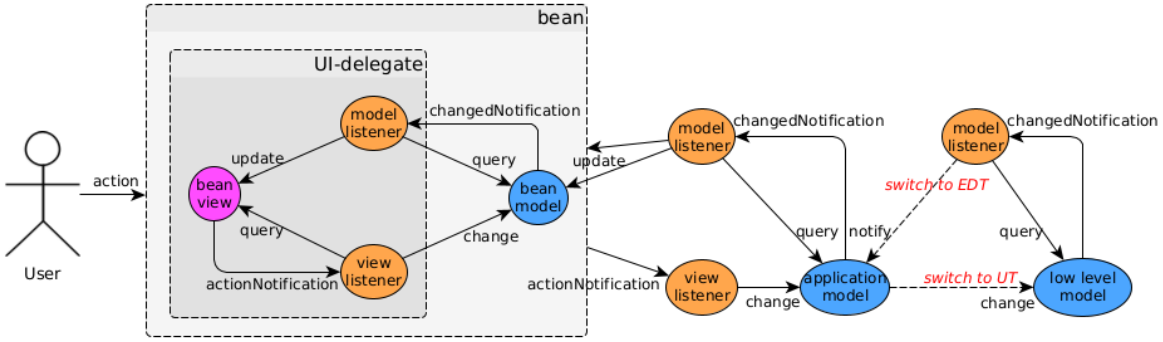

FIGURE 7 – Collaboration entre les éléments de l'architecture MVC supportant la concurrence (modèle complexe).

Il y a maintenant trois modèles en jeu :

— le modèle de l'application (au centre sur le schéma), qui définit les commandes métier de l'application, les services en quelque sorte (par exemple, dans un éditeur de texte, « ouvrir un fichier texte pour l'édition ») ;

— le modèle bas-niveau (à droite), qui définit les commandes d'interaction avec le système (par exemple, « charger le contenu du fichier en mémoire », ou plus généralement la couche d'accès aux données) ;

— le modèle de l'interface graphique de l'application (à gauche), qui définit l'état sous-jacent aux composants de l'interface graphique (par exemple, le `Document` utilisé par la zone de texte qui affiche le texte en cours d'édition).

Le modèle de l'application délègue l'exécution des calculs longs, ou pouvant bloquer le fil courant d'exécution, au modèle bas-niveau à l'aide de messages asynchrones, c'est-à-dire exécutés sur des threads utilisateurs, afin de ne pas accaparer EDT.

Ensuite, le modèle bas-niveau travaille sur un ou plusieurs threads utilisateurs. Son fonctionnement est observé (au sens du patron de conception OBSERVER) par un écouteur préalablement défini à l'aide d'une classe locale anonyme dans le modèle de l'application. Cet écouteur a, de plus, la charge de rétablir le flot de contrôle sur EDT, ce qui est absolument nécessaire puisque seul EDT a le droit de mettre à jour les composants graphiques.

Enfin, le modèle de l'interface graphique est parfois dispersé au sein des composants graphiques qui constituent la partie « vue » de l'application, comme suggéré sur la figure 7.

Mais on peut préférer centraliser dans un seul objet toutes les informations nécessaires à la vue, en regroupant une partie des modèles de certains composants graphiques (complexes) ainsi que des informations du genre « tel bouton doit être activable ou non ». Cette façon de faire peut grandement simplifier le traitement des mises à jour de la vue.

Une dernière remarque, de taille, avant de terminer cette section. Le lecteur attentif aura deviné que la récupération des informations du modèle bas-niveau par celui de l'application peut être critique puisque ces deux modèles sont manipulés sur deux threads distincts. Or si le mécanisme de notification entre les deux modèles est de type *push* il n'y a pas lieu de prendre des précautions particulières car toutes les données transitent entre le modèle bas-niveau et son écouteur sous forme d'événements, donc par l'intermédiaire de paramètres plutôt que par des variables partagées. En revanche, si le mécanisme choisi est de type *pull*, il faut s'assurer que la communication entre les modèles met en place un échange sûr de données cohérentes. Par exemple, il faut récupérer les données du modèle bas-niveau pendant que l'on est sur le thread utilisateur et avant de basculer sur EDT. La technique exposée, par l'écoute du changement de valeurs de propriétés du modèle bas-niveau, permet au développeur de s'affranchir de toute synchronisation et simplifie ainsi grandement le développement de l'application.

5 Conclusion

Après avoir rappelé ce qu'était l'architecture MVC et la manière dont elle pouvait être implantée en Java, nous avons présenté l'architecture dite « à modèle séparable » des composants de la bibliothèque Swing. Nous avons expliqué que cette dernière forme était une adaptation de la première, permettant de minimiser l'interface entre les parties vue et contrôleur de ces composants.

L'utilisation de la bibliothèque Swing impose une contrainte forte : tout composant graphique et son modèle doivent être manipulés sur EDT (car *Swing is not thread-safe*). Ceci impose que l'activité globale de l'application se déroule sur EDT, et il semblerait que l'on tombe alors dans un cercle vicieux : l'activité du modèle se déroule sur EDT car elle trouve son origine dans l'action de l'utilisateur, or si cette activité est trop longue il faut qu'elle se déroule sur un autre thread que EDT et, si le modèle notifie ses vues, certains composants Swing ne seront pas mis à jour sur EDT !

Nous avons alors proposé une méthode qui permet au développeur, tout en respectant la contrainte engendrée par l'utilisation de la bibliothèque Swing (modification des composants graphiques sur EDT), d'utiliser autant de threads que nécessaire à la bonne exécution de son application. Il suffit pour cela de découper le modèle en plusieurs parties (entre une et trois selon la complexité du modèle de l'application) et nous avons montré comment exécuter, d'une part les tâches longues (sur des threads utilisateur), et d'autres part les manipulations des composants graphiques et de leurs modèles (sur EDT systématiquement). L'usage de cette technique a l'avantage de permettre l'échange de données sans synchronisation entre les différents modèles.

Références

[1] Steeve Burbeck. Applications programming in smalltalk-80(TM) : How to use Model-View-Controller (MVC). https://www.researchgate.net/publication/238719652_Applications_programming_in_smalltalk-80_how_to_use_model-view-controller_mvc, 1992.

[2] Robert Eckstein. Java SE application design with MVC. http://www.oracle.com/technetwork/articles/javase/mvc-136693.html, 2007. (Avec les images : http://browse.feedreader.com/c/The_skiing_cube/251397735).

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : elements of reusable object-oriented software*. Addison-Wesley, 1994.

[4] Graham Hamilton. Multithreaded toolkits : A failed dream ? https://community.oracle.com/blogs/kgh/2004/10/19/multithreaded-toolkits-failed-dream, 2004. (Archivé ici : https://titanwolf.org/Network/Articles/Article?AID=f11696d7-ca6f-44f4-82ef-e4cdc0b264e5).

[5] Trygve Reenskaug. The original MVC reports. http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf, 1979.