

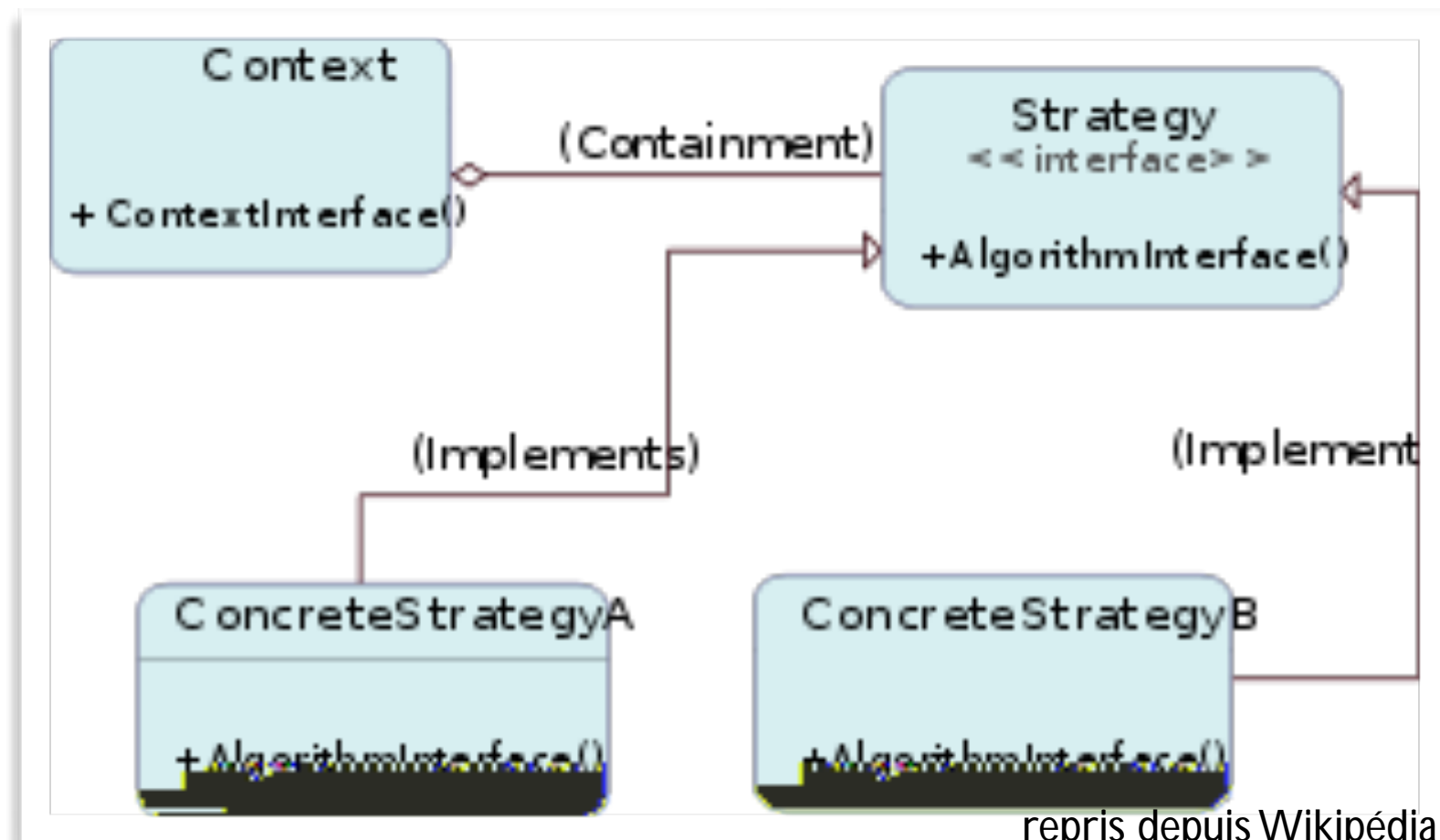
Interfaces Graphiques

Jean-Baptiste.Yunes@univ-paris-diderot.fr

Université Paris Diderot

©2014

- Rappel
 - un Layout est un *composant* du Strategy Pattern
 - sélection dynamique d'un algorithme
 - cet algorithme est chargé du placement des composants dans un container

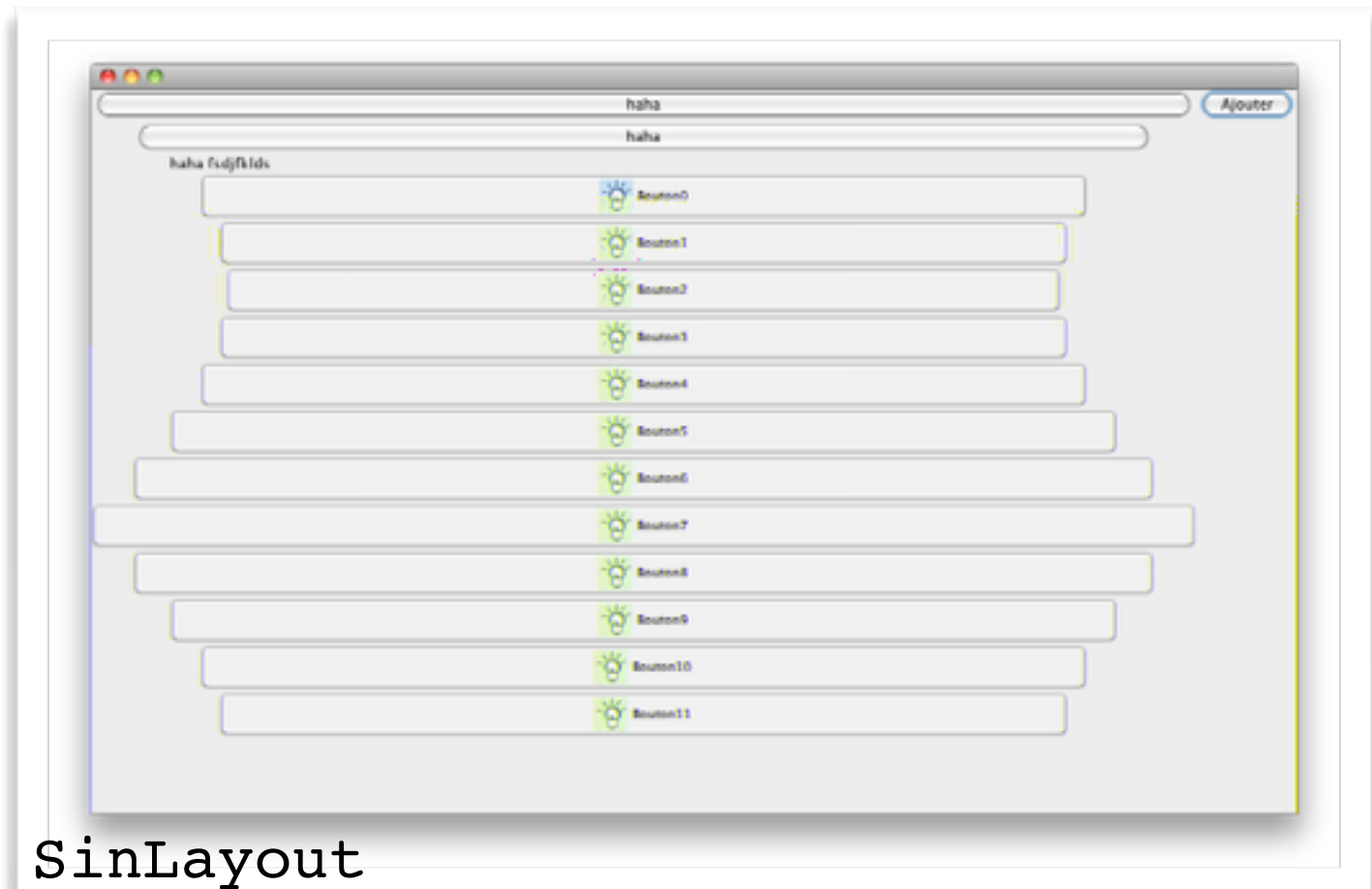
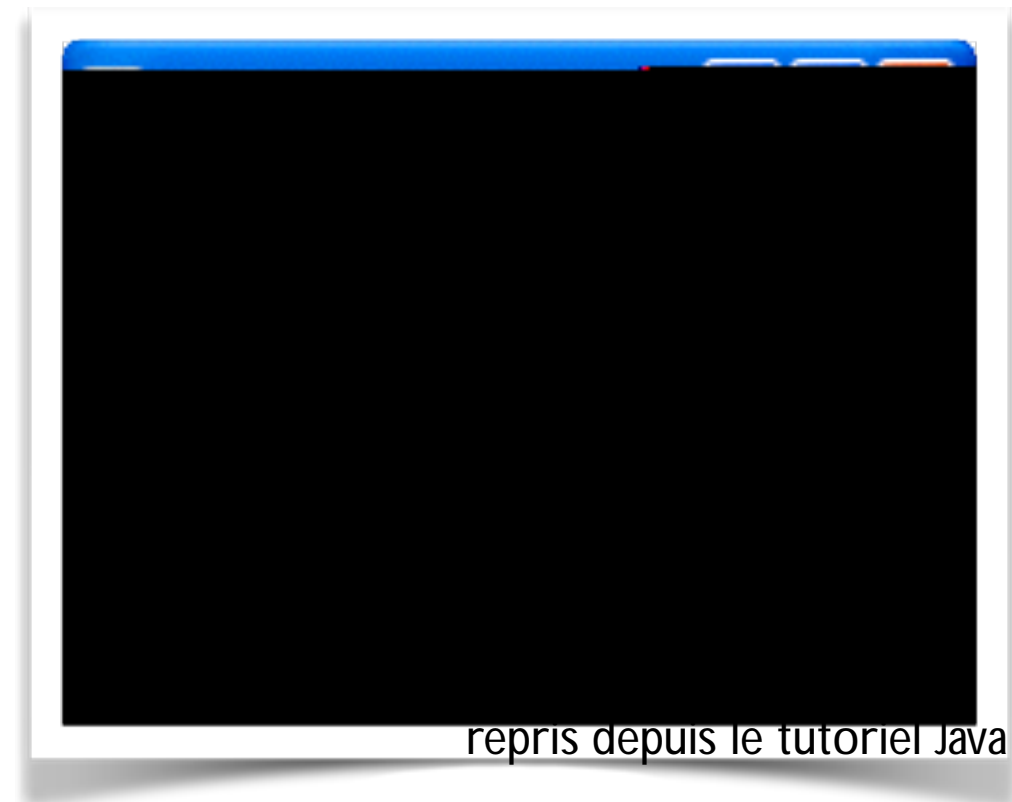


- l'interface `java.awt.LayoutManager`
 - `void addLayoutComponent(String, Component)`
 - gestionnaire annexe de la collection
(ex : `BorderLayout`)
 - `void removeLayoutComponent(Component)`
 - quoi faire quand un composant est enlevé
- `Dimension`
`preferredLayoutSize(Container)`
 - est appelée lorsque il est nécessaire de déterminer la taille *idéale* du container
(ex. : `pack()`).

- l'interface `java.awt.LayoutManager`
 - `Dimension minimumLayoutSize(Container)`
 - est appelée à différents moments lorsqu'il est nécessaire de déterminer la taille minimale requise pour afficher le `container`
 - `void layoutContainer(Container)`
 - méthode *principale* des layout, dans laquelle est effectivement déterminée la taille et la position des composants contenus dans le `container`, lequel les interrogent et les placent.

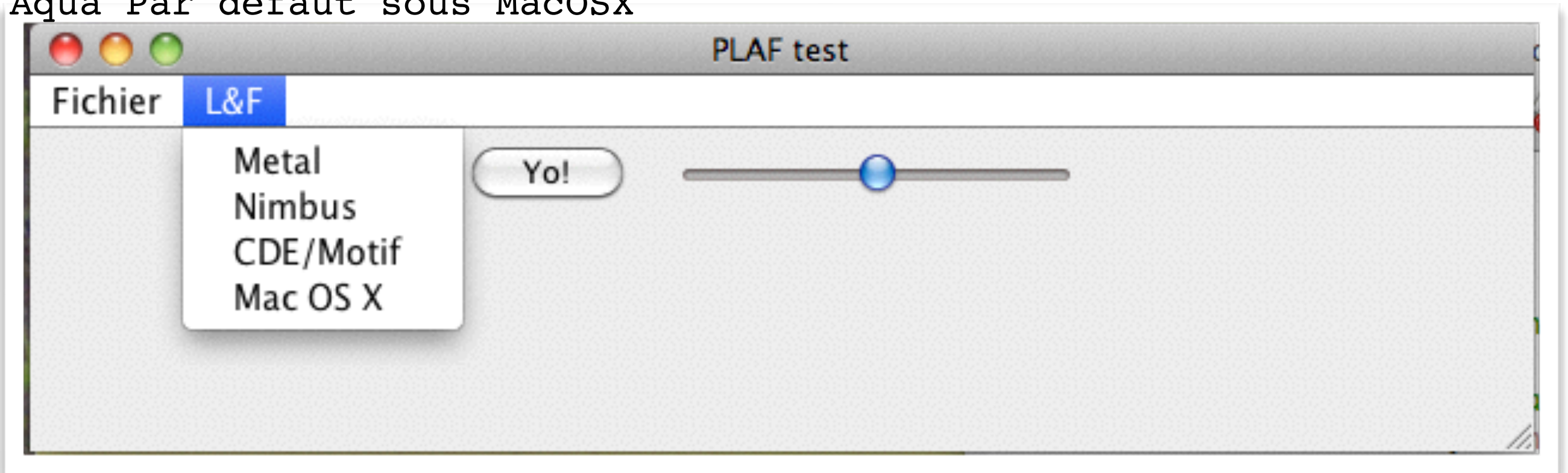
- l'interface `LayoutManager2` fournit d'autres méthodes :
 - `void addLayoutComponent(Component, Object);`
 - `float getLayoutAlignmentX(Container)`
 - `float getLayoutAlignmentY(Container)`
 - `void invalidateLayout(Container)`
 - `Dimension maximumLayoutSize(Container)`

- deux exemples :
 - tutoriel Java :
`DiagonalLayout`
 - code source
fourni
- exemples du cours :
`SinLayout`
- disponible depuis
la page web du
cours

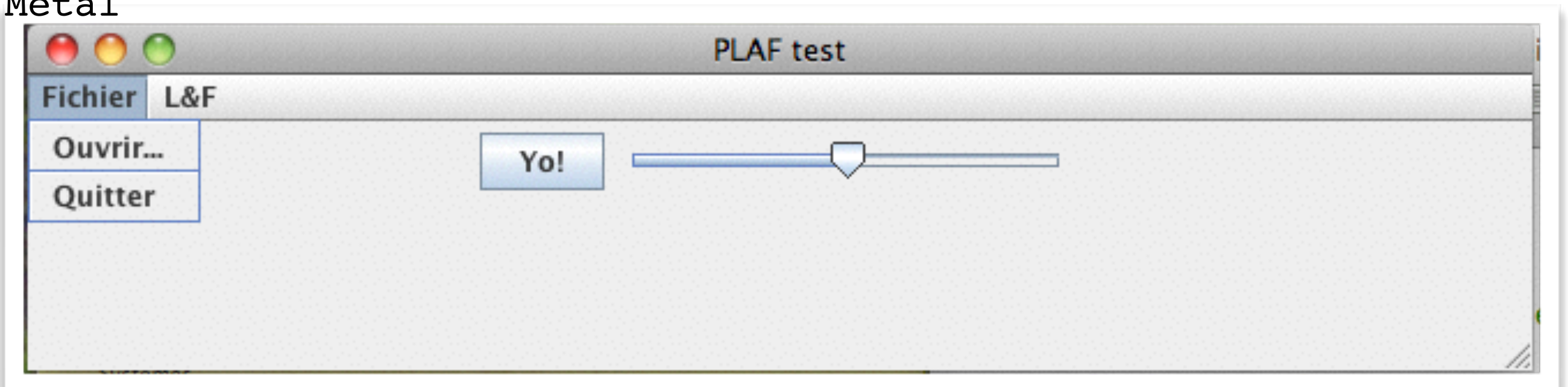


Les apparences de MacOSX Snow Leopard

Aqua Par défaut sous MacOSX

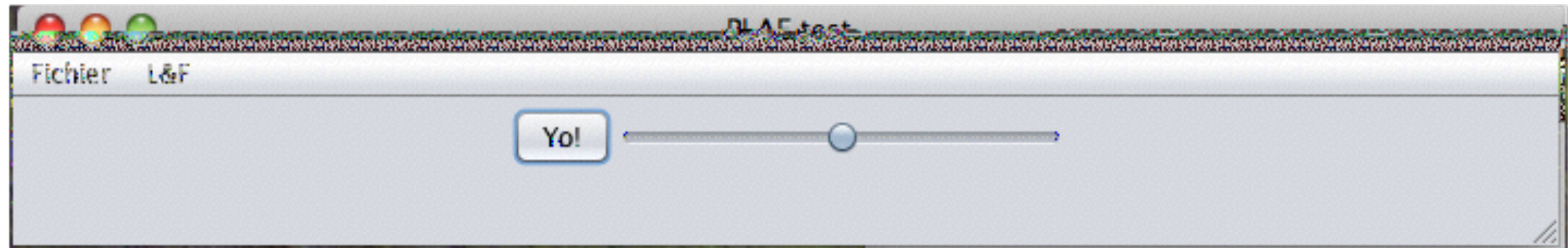


Metal

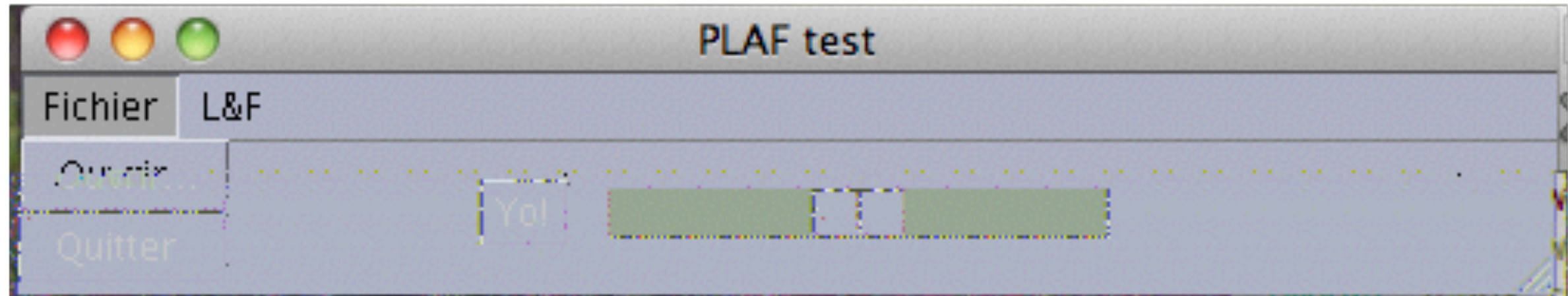


Les apparences de MacOSX Snow Leopard

Nimbus



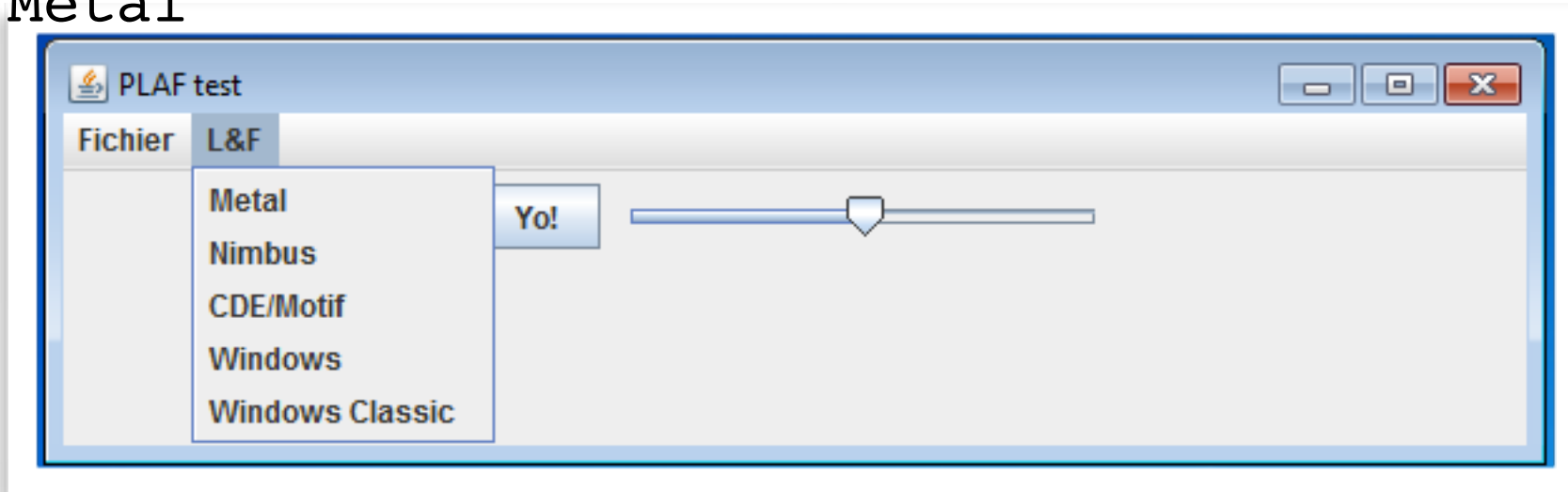
CDE/Motif



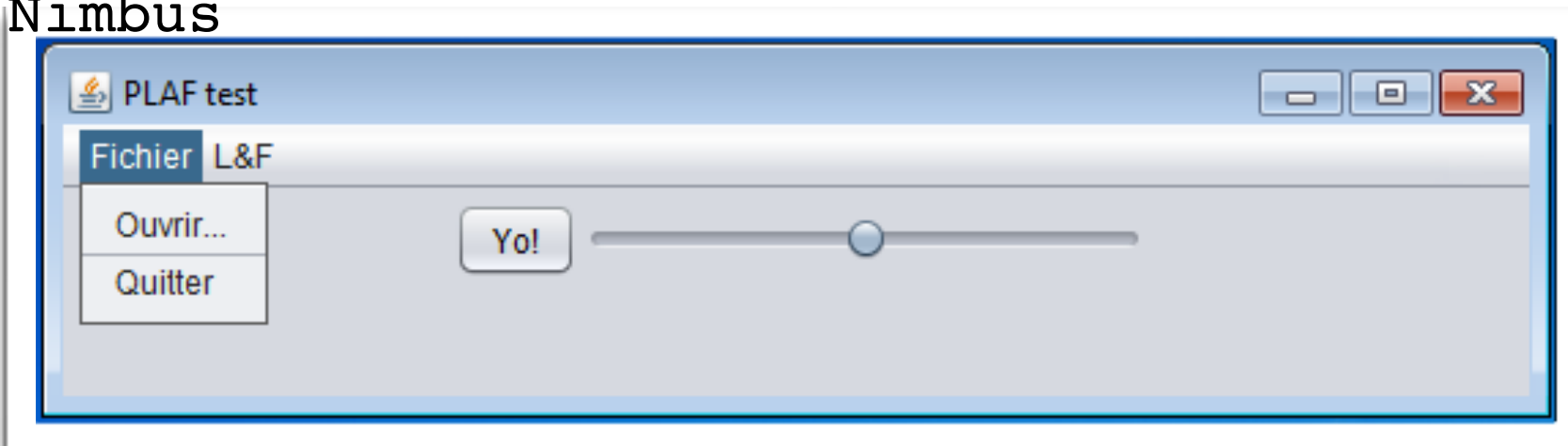
Look and Feel

Les apparences sous Windows 7

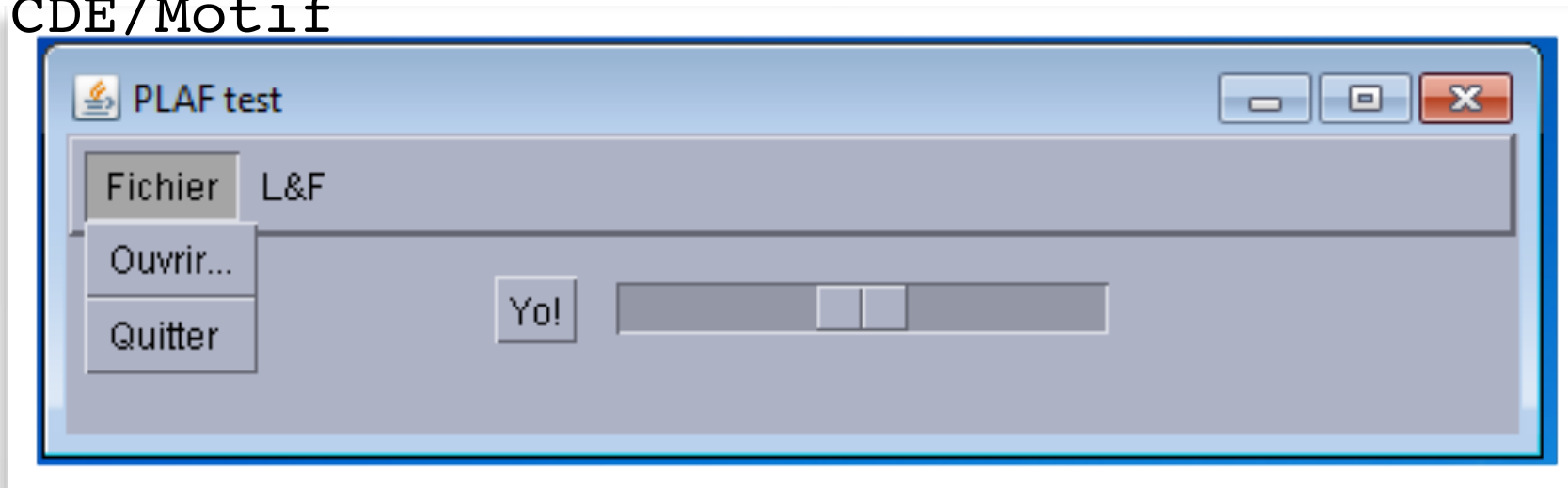
Metal



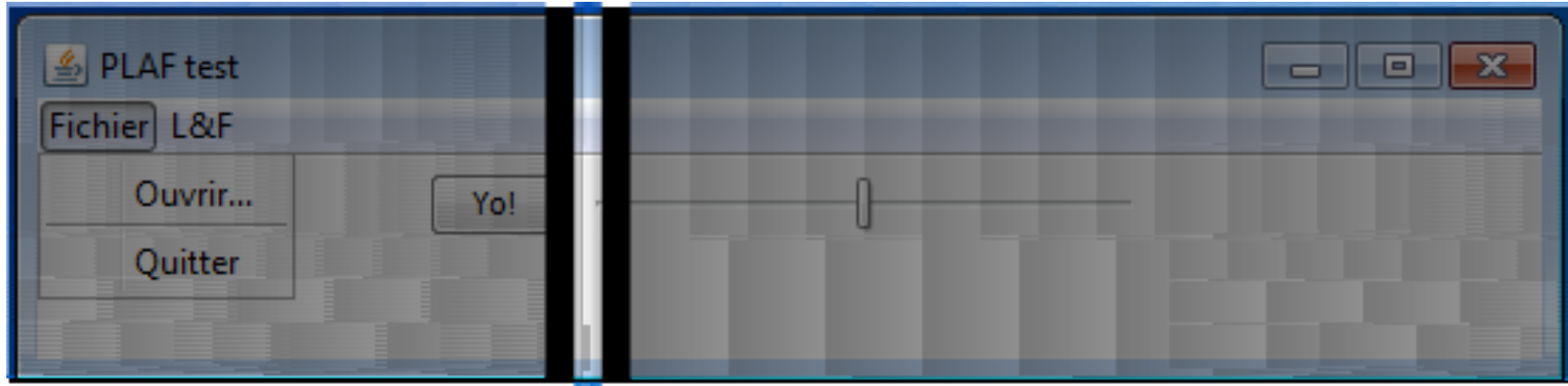
Nimbus



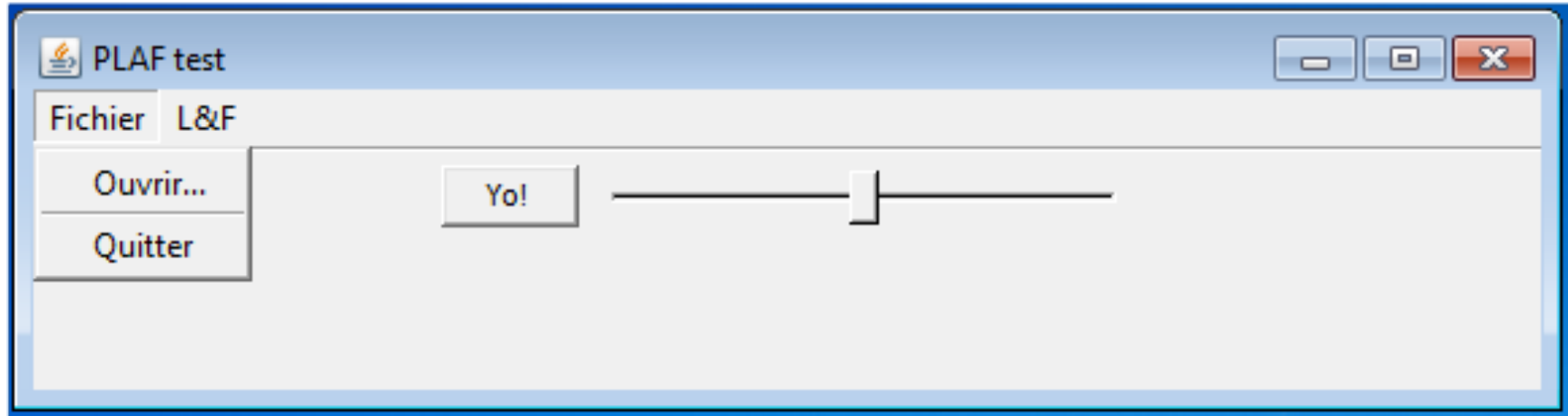
CDE/Motif



7



Classic



- La classe `UIManager` propose des services utilitaires de gestion des Apparences (Look-and-Feels)
- apparences disponibles
 - `UIManager.LookAndFeelInfo [] getInstalledLookAndFeels();`
- modification de l'apparence courante
 - `setCurrentLookAndFeel(LookAndFeel)`
- etc.

- `UIManager.LookAndFeelInfo` :
 - décrit une apparence
 - `getName()` ;
 - un nom court à employer pour représenter l'apparence dans une interface
 - `getClassName()` ;
 - le nom complet à employer pour sélectionner une apparence

- Si la modification est opérée dynamiquement
 - demander à l'interface de se remettre entièrement au goût du jour
 - `SwingUtilities.updateComponentTreeUI (Component) ;`
 - éventuellement `pack () ...`
- Exemple : `PLAF.java`

- L'apparence `Metal` autorise la création de variations
- Thèmes
 - `DefaultMetalTheme`, `OceanTheme`
 - à sous-classer et modifier pour obtenir une variante...
 - `MetalLookAndFeel.setCurrentTheme(MetalTheme)`
- Exemple : `PLAFTheme.java`

- La création d'une apparence s'obtient :
 - par programmation
 - par une définition externe (XML)

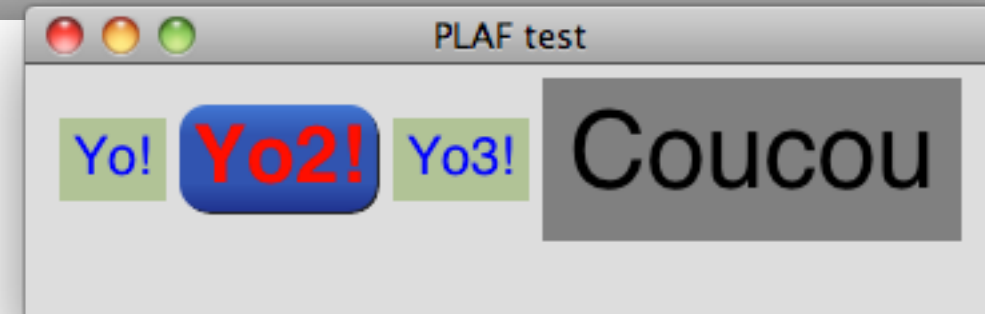
- Technique XML :
 - créer une instance de `SynthLookAndFeel`
 - lui associer une description XML :
 - `load(InputStream, Class<?>) ;`
 - le premier paramètre est le flux de lecture du fichier XML
 - le second paramètre servira à déterminer comment charger des ressources associées (`ClassLoader`, etc.)

- les composants appartiennent tous à une classe et l'on peut définir une apparence pour toutes les instances d'une classe donnée
- mais, les composants peuvent être nommés *via*
 - `setName(String);`
 - ce qui autorise à définir certaines particularités pour certaines instances particulières (boutons de dialogues, etc)
- mécanisme similaire à class/id de HTML

- la balise `<style id="identificateur">...</style>`
 - permet de définir des caractéristiques d'apparence
- la balise `<bind style="id" type="name|region" value="exp" />`
 - permet d'associer le style d'identité *id* aux objets correspondants à l'expression
 - `name` : objets nommés *exp*
 - `region` : objets génériquement nommés *exp*

Création d'un Look and Feel via XML

```
<style id="yo2Style">
  <!--opaque value="TRUE"/-->
  <property key="Button.textShiftOffset" type="integer"
    value="1"/>
  <font name="Helvetica-Bold" size="30"/>
  <insets top="5" left="5" right="5" bottom="5"/>
  <state>
    <imagePainter method="buttonBackground" path="button.png"
      sourceInsets="20 20 20 20"/>
    <color value="GREEN" type="BACKGROUND"/>
    <color value="RED" type="TEXT_FOREGROUND"/>
  </state>
  <state value="PRESSED">
    <imagePainter method="buttonBackground" path="button3.png"
      sourceInsets="20 20 20 20"/>
    <color value="GREEN" type="BACKGROUND"/>
    <color value="#FF6666" type="TEXT_FOREGROUND"/>
  </state>
</style>
<bind style="yo2Style" type="name" key="Y02"/>
```



- Un exemple : `PLAFSynth.java`

- *A Swing Architecture Overview*

The Inside Story on JFC Component Design

par Amy Fowler

- Pour personnaliser un composant il est nécessaire de prendre en préoccupation :
 - du rendu
 - un délégué de type `ComponentUI`
 - de la logique *métier*
 - un modèle
 - des services de base
 - `JComponent` + export partiel de la logique métier

- la plomberie du rendu
 - rappel : le rendu Swing est effectué non pas par le composant lui-même mais par délégation
- la classe `JComponent` fournit les services
 - `ComponentUI` `getUI()` / `setUI(ComponentUI)`
 - `updateUI()`
 - permet de capturer le moment où le `LookAndFeel` doit être pris en compte

- un `JComponent` ne se dessine pas lui-même
- sinon comment pourrait-il prendre en compte une nouvelle apparence ?
- emploi du motif conceptuel de la Délégation
 - à chaque `JComponent` est associé un objet s'occupant du rendu du composant
 - `ComponentUI`

- à un `JButton` est associé un `ButtonUI`
- à un `JLabel` est associé un `LabelUI`
- etc.
- modifiable par appel à
 - `setUI (ComponentUI)`
- consultable par appel à
 - `getUI (ComponentUI)`

- Un exemple : `PLAFUI.java`

- La logique métier est normalement (MVC) encapsulée dans une instance de Modèle
 - définir son propre modèle
- il en existe de prédéfinis
 - ButtonModel
 - ListModel
 - etc.

- Exporter quelques services du modèle afin de rendre la vie plus facile
- et après tout, est-ce que le programmeur doit se préoccuper de ces histoires de modèle ?

- Un exemple : `Speedometer.java`