SwingStates

Nicolas Roussel

http://mjolnir.lille.inria.fr/~roussel/

mailto:nicolas.roussel@inria.fr

Transparents adaptés de ceux de Caroline Appert (CNRS) et Michel Beaudouin-Lafon (Univ. Paris-Sud)

SwingStates ?

Une librairie crée en 2006 au LRI
par Caroline Appert et Michel Beaudouin-Lafon

Décrite dans plusieurs publications, dont

C. Appert and M. Beaudouin-Lafon. “SwingStates: adding state machines to Java and the Swing toolkit”. *Software Practice and Experience*. 38 (11), p. 1149-1182. September 2008.

C. Appert and M. Beaudouin-Lafon. “SMCanvas : augmenter la boîte à outils Java Swing pour prototyper des techniques d'interaction avancées”. In Actes d’IHM 2006, p. 99-106. ACM, avril 2006.

Disponible gratuitement sous licence LGPL

http://swingstates.sourceforge.net/

https://sourceforge.net/projects/swingstates/

Pourquoi SwingStates ?

L'interaction se programme en Java/Swing par envoi de messages, avec des *listeners*

Les classes internes anonymes de Java facilitent la chose

```
final JButton bouton = new JButton( "monBouton" ) ;
bouton.addActionListener( new ActionListener() {
 public void actionPerformed( ActionEvent e ) {
      someObject.doSomething() ;
 }
}) ;
```

mais l'enchaînement des actions liées à une interaction est difficile à lire, difficile à comprendre, difficile à maintenir

Qu’apporte SwingStates ?

SwingStates permet de programmer l’interaction en la décrivant à l’aide de machines à états

- les machines sont spécifiées en Java, au milieu du reste du code
- elles peuvent ensuite être attachées à un widget

SwingStates fournit un canevas pour le dessin 2D

- les formes à afficher sont ajoutées une à une au canevas
- pour chacune d’elle, on peut spécifier la géométrie, des attributs graphiques, un parent et des tags
- SwingStates s’occupe de l’affichage et de la sélection (*picking*), des animations

SwingStates intègre aussi deux reconnaisseurs de gestes, mais c’est une autre histoire…

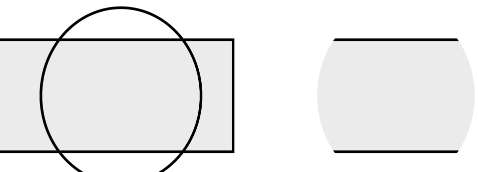
Dessin structuré sur un Canvas

Dessin structuré sur un Canvas

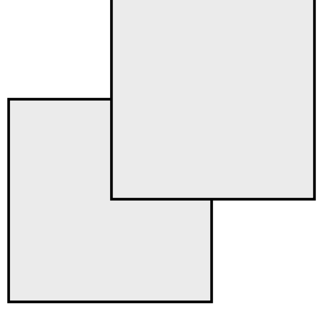
Les formes peuvent être de 5 types

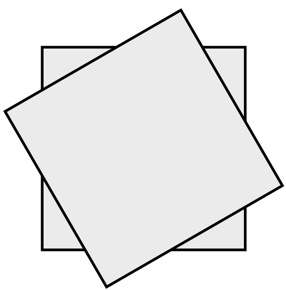
- `CRectangle`, `CEllipse` : des formes simples
- `CPolyline` : une forme avec un contour arbitraire
- `CText` : une chaîne de caractères
- `CImage` : une image bitmap
- `CWidget` : un widget Swing

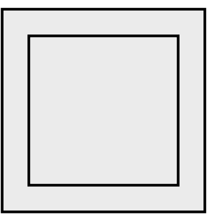
Les formes peuvent être *clippées*



Des transformations affines peuvent leur être appliquées (translation, rotation et changement d'échelle)







Attributs graphiques des formes

Pour les formes géométriques

- affichage ou non de l'intérieur et de la bordure
- couleur du fond (`Paint` de Java2D)
- couleur et aspect de la bordure (`Paint` et `Stroke` de Java2D)
- opacité

Pour le texte

- police de caractère (`Font` de Java2D)
- couleur (`Paint` de Java2D)
- opacité

Pour les images

- source (e.g. un fichier)
- opacité

Ordre du dessin et de la sélection, hiérarchie de formes

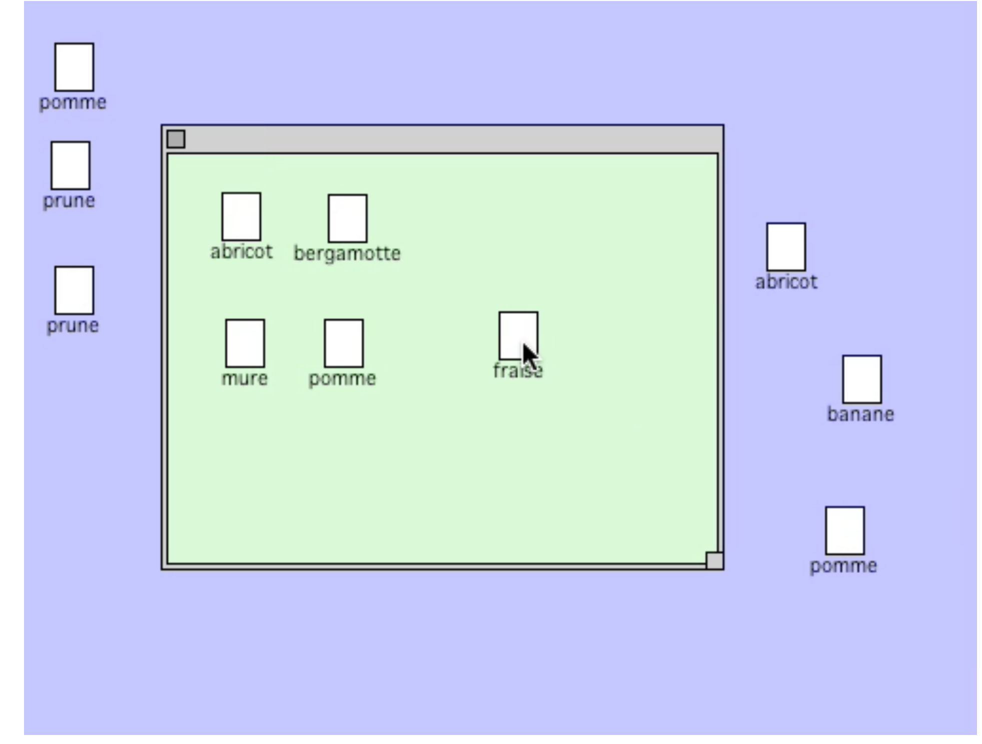
Les formes sont dessinées dans l'ordre de la liste du `Canvas`

La sélection se fait en ordre inverse, pour respecter la superposition

Pour chaque forme, on peut choisir si elle doit être dessinée et si elle peut être sélectionnée

Toute forme peut être attachée à un *parent*

Les coordonnées d'une forme attachées sont spécifiées dans le repère du parent



Syntaxe chaînée

Les méthodes de SwingStates renvoient le plus souvent l'objet sur lequel elles ont été appliquées (`this`)

Ceci permet de chaîner les appels

```
shape.setFillColor(Color.RED).translateBy(10, 100)

canvas.addShape(shape1).addShape(shape2)
```

Les tags

Chaque forme peut porter un nombre arbitraire de tags, et un même tag peut être mis sur un nombre arbitraire de formes

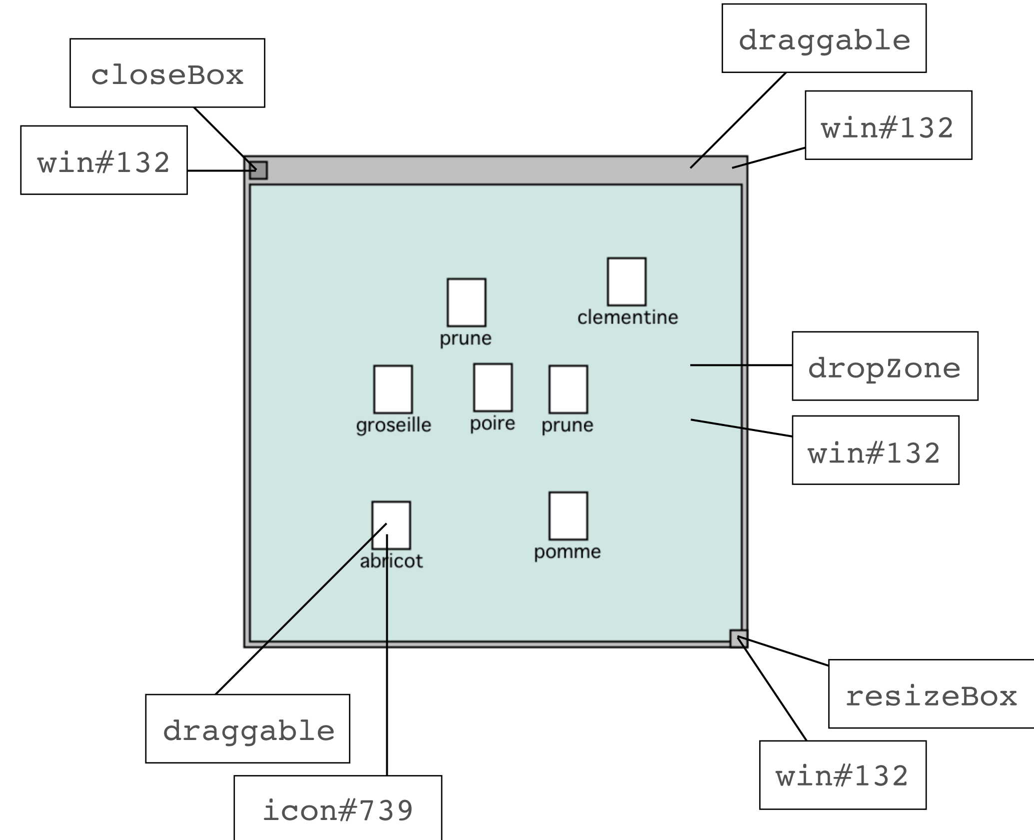
Un tag peut être utilisé pour marquer l'appartenance à une classe d'objets (e.g. "closeBox", "movableShape") ou à une structure particulière (e.g. "window#162") par exemple

La plupart des opérations applicables à une forme peuvent s'appliquer à un tag, i.e. à l'ensemble des formes qui le portent

Les tags (suite)

Un tag permet donc de nommer un ensemble d'objets, de les manipuler ensemble et de définir des interactions qui leur seront applicables

Exemple : une fenêtre, ses décorations/interacteurs, des icônes



Les tags (suite)

Deux types de tags sont possibles

Les tags extensionnels (`CExtensionalTag`)

- sont mis explicitement sur les formes (`addTag`, `removeTag`), ce qui déclenche l'appel des méthodes `added` et `removed` du tag
- le plus souvent, ce sont des `CNamedTag` définis par un simple nom

Les tags intentionnels (`CIntentionalTag`)

- sont définis dans une sous-classe de `CIntentionalTag` par un prédicat implémenté dans la méthode `criterion`
- à chaque utilisation d'un tag intentionnel, le prédicat est testé sur l'ensemble des formes du `Canvas` (ce qui est potentiellement couteux…)
- un tag intentionnel utile : `CHierarchyTag`, qui permet de manipuler tous les descendants d'une forme

Les tags (suite)

Exemple de tag extensionnel

```
CExtensionalTag selectionTag = new CExtensionalTag() {
  public void added(CShape s){ s.setFillPaint(Color.GREEN); }
  public void removed(CShape s){ s.setFillPaint(Color.LIGHT_GRAY); }
};
```

Exemple de tag intentionnel

```
CIntentionalTag smallShapeTag = new CIntentionalTag {
  public boolean criterion(CShape s) {
      return (s.getWidth()<500 && s.getHeight()<500) ;
  }
}
```

Animations

Certains attributs géométriques et graphiques peuvent être animés (couleur de bord et de fond, opacité, position, taille, orientation)

Exemple

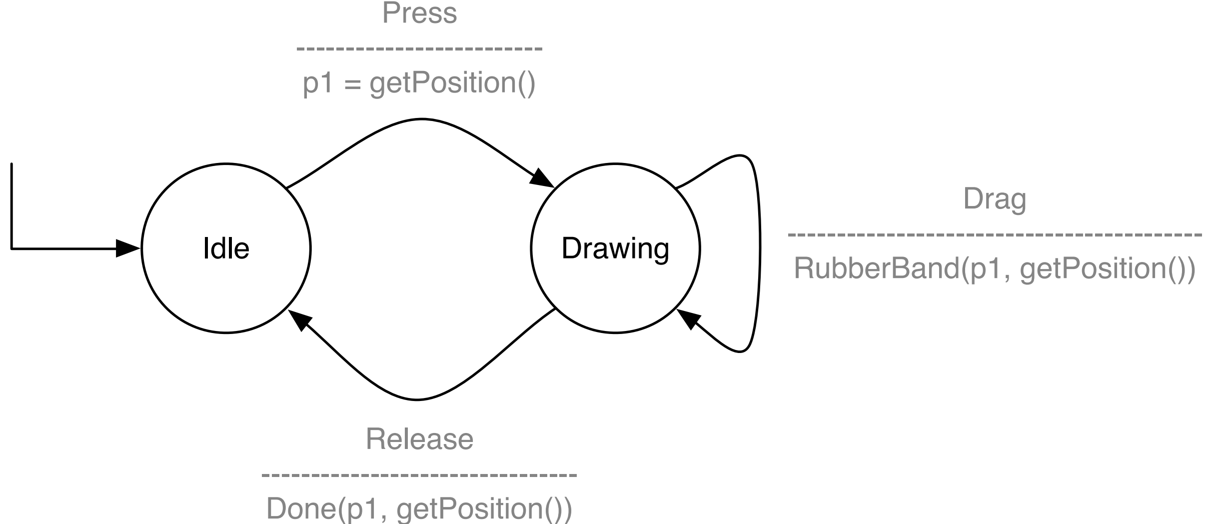
```
rect = canvas.newRectangle(100, 150, 1, 1) ;
animation = new AnimationScaleTo(400, 600) ;
rect.animate(animation) ;
// l'animation commence immédiatement
```

D'autre animations peuvent être définies en étendant la classe de base `Animation`

Machines à états

Exemple simple : tracé de ligne élastique (rappel)

Sous forme graphique



On aimerait pouvoir écrire quelque chose du genre

```
Etat “Idle” {
   Transition sur Press => “Drawing” { p1 = getPosition() }
}
Etat “Drawing” {
   Transition sur Move => “Drawing” { RubberBand(p1, getPosition()) }
   Transition sur Release => “Idle” { Done(p1, getPosition()) }
}
```

Avec SwingStates

SwingStates s'appuie aussi sur les classes anonymes internes

```
CStateMachine mach = new CStateMachine() {

     private CSegment seg ;
     private Point2D p1 ;

     public State idle = new State() {

          Transition startDrawing = new Press(BUTTON1, ">> drawing") {
               public void action() {
                    p1 = getPoint() ;
                    seg = ((Canvas)getEvent().getSource()).newSegment(p1, p1) ;
               }
          } ;

     } ;

     public State drawing = new State() {

          Transition stillDrawing = new Drag(BUTTON1) {
               public void action() {
                    seg.setPoints(p1, getPoint()) ;
               }
          } ;

          Transition doneDrawing = new Release(BUTTON1, ">> idle") ;

     } ;

}
```

Le premier état est l'état initial

L'événement peut être paramétré

Sucre syntaxique…

Accès aux champs et méthodes de l'état et de la machine

Les transitions sont testées dans l'ordre de leur déclaration

Les états doivent être publics

Press, Release, Drag et d'autres événements, éventuellement paramétrés par OnShape ou OnTag

OnTag > OnShape > ∅

Le reste de l'application

```
import fr.lri.swingstates.canvas.Canvas ;
import fr.lri.swingstates.canvas.CStateMachine ;
import fr.lri.swingstates.canvas.CSegment ;

import javax.swing.JFrame ;
import java.awt.geom.Point2D ;

class RubberBandDemo {

    static public void main(String[] args) {

        Canvas canvas = new Canvas(400,400) ;
        CStateMachine mach = new CStateMachine() { … } ;
        mach.attachTo(canvas) ;

        JFrame frame = new JFrame() ;
        frame.getContentPane().add(canvas) ;
        frame.pack() ;
        frame.setVisible(true) ;

   }

}
```

Si on veut voir la machine à états

```
...
import fr.lri.swingstates.debug.StateMachineVisualization ;
...

class RubberBandDemo {

    static public void main(String[] args) {

        ...

        JFrame viz = new JFrame() ;
        viz.getContentPane().add(new StateMachineVisualization(mach)) ;
        viz.pack() ;
        viz.setVisible(true) ;

    }

}
```

L'état courant et la dernière transition déclenchée sont colorés

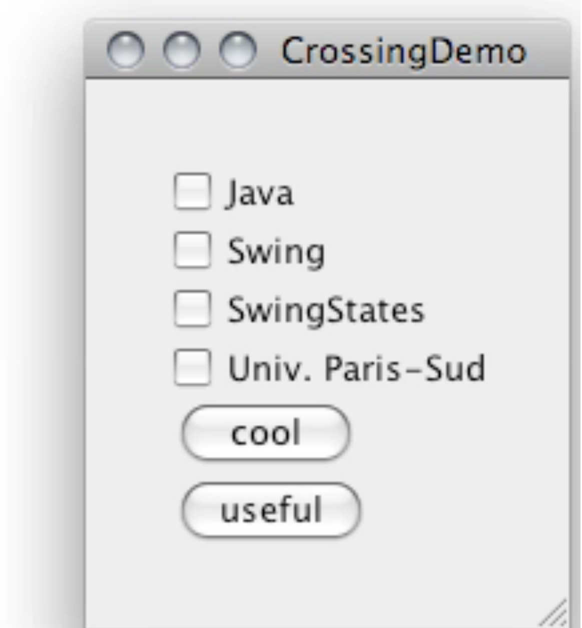
Les machines ne sont pas limitées aux Canvas SwingStates

La classe `JStateMachine` permet de redéfinir l'interaction avec un widget Swing quelconque

Les transitions ont des noms du type `*OnComponent`

La machine peut être attachée

- à un widget particulier
- à tous les widgets d'une classe particulière
- à tous les widgets qui ont un tag particulier (via les classes `JTag`, `JNamedTag`)



Comment concevoir les machines à états ?

A quel niveau les placer ?
Comment éviter l'explosion du nombre d'états ?

Pour des tâches indépendantes, on peut faire tourner des machines en parallèle (e.g. sélection de commande / tooltips)

Les machines peuvent aussi communiquer vers l'extérieur

- `fireEvent` permet à une machine d'émettre un événement
- `addStateMachineListener` permet à un listener passé en paramètre de recevoir les événements émis par une machine