SwingStates
programmation d'interactions graphiques en Java

Michel Beaudouin-Lafon - mbl@lri.fr
Caroline Appert - appert@lri.fr

http://swingstates.sourceforge.net

Programmer l'interaction en Java

"Listeners"

- implémentation en Java d'une fonction de rappel
- un objet "listener" est associé à un widget
- une méthode de ce "listener" est appelée lorsque le widget change d'état

Avantages

- concept simple
- notation facilitée avec les classes internes anonymes

Inconvénients

- une interaction complexe est répartie dans plusieurs listeners
- la logique de l'interaction est difficile à suivre
- maintenance difficile

Alternative : les machines à états

Machines à états

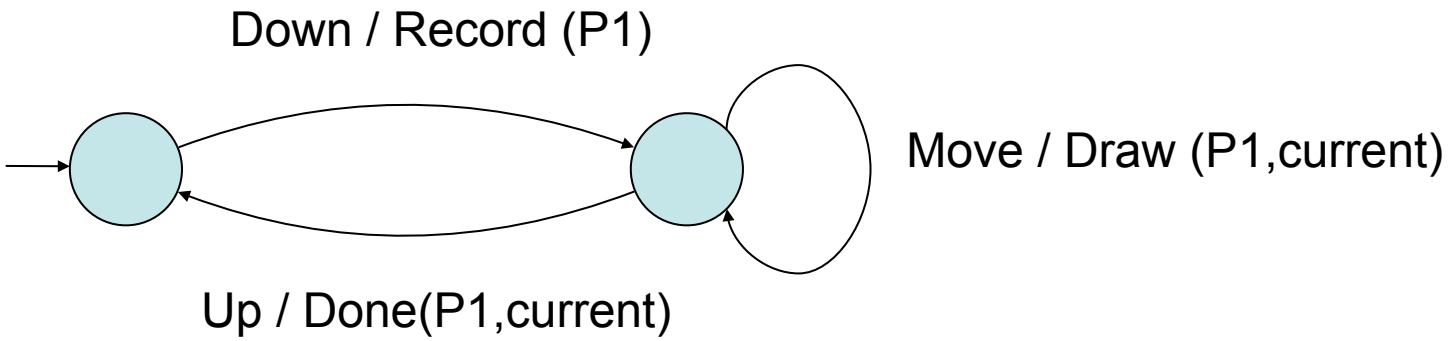
Automates à états fini

- Etat (cercle) = état de l'interaction
- Transition (flêche) = événements d'entrée (Up, Down, Move, ...)

Machine à états

- actions associées aux transitions (après le "/")
- conditions associées aux transitions (après le "&")

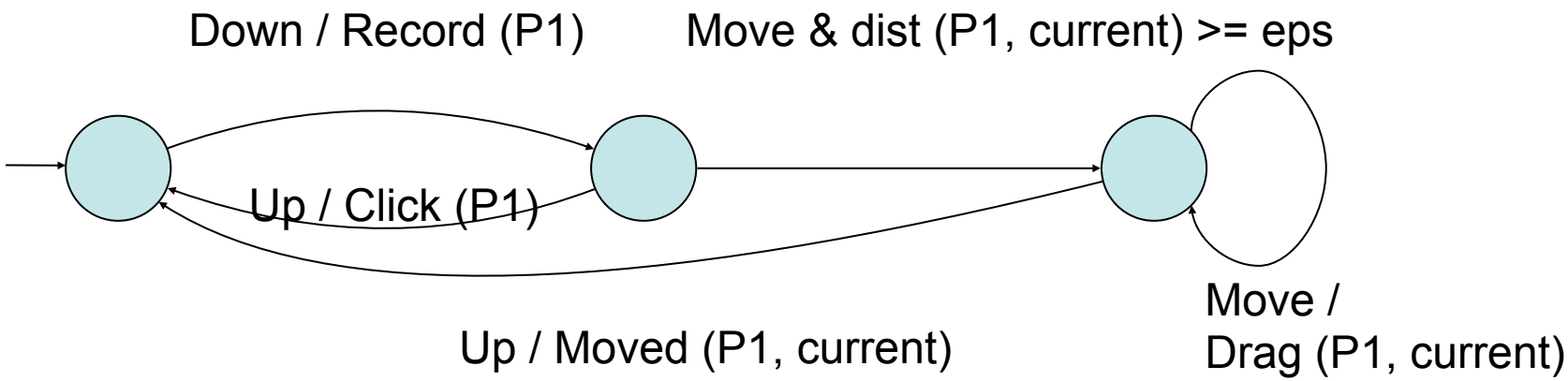
Exemple : "Rubber-band"



Machines à états

Combiner sélection et tracé

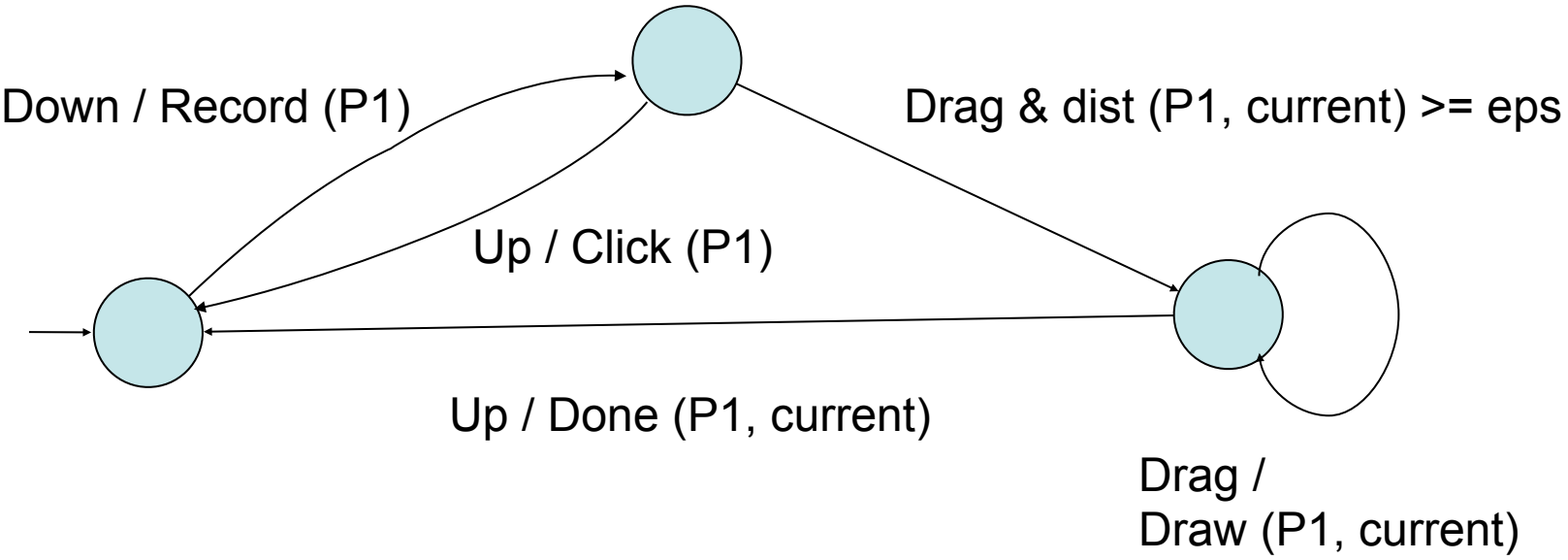
- Hystéresis : le tracé ne démarre qu'après un déplacement suffisant



Machines à états

Combiner sélection et tracé

- Hystéresis : le tracé ne démarre qu'après un déplacement suffisant



SwingStates : principes de base

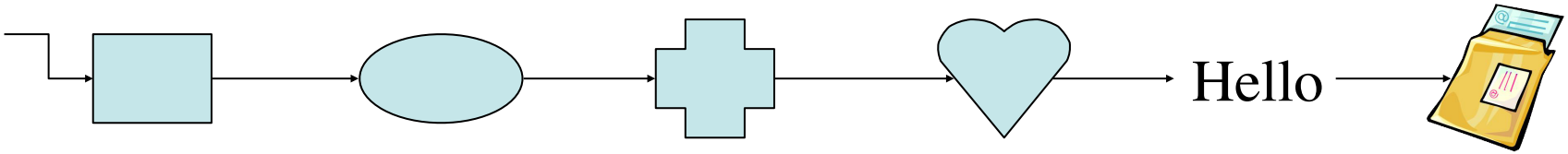
Dessin structuré : widget Canvas

- liste d'affichage de formes graphiques
- formes graphiques définies par
  - une géométrie
  - des attributs graphiques
  - des "tags"
- gestion de l'affichage et du "picking"

Machines à états

- spécification directement en Java des états et des transitions
- association à un Canvas
- association à un widget Swing quelconque

Widget Canvas : dessin structuré



Liste d'affichage :

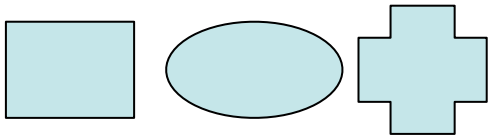
Affichage dans l'ordre de la liste : le premier objet est sous les autres

"Picking" dans l'ordre inverse pour respecter l'ordre de superpostion

Géométrie :

CPolyline : forme quelconque

CRectangle, CEllipse : formes simples

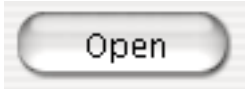


CText : chaîne de caractères

Hello

CImage : image

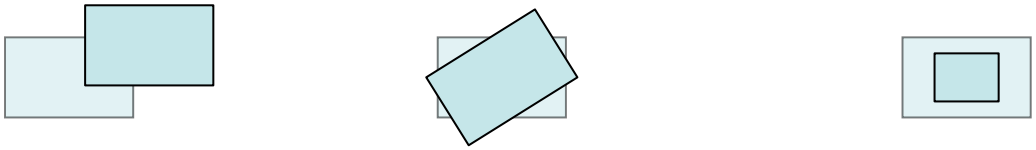
CWidget : widget Swing



Dessin structuré : géométrie

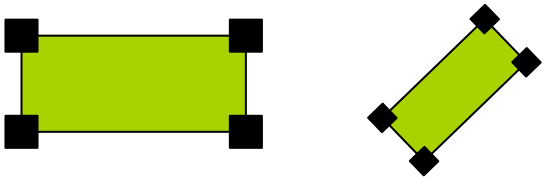
- transformations affines

  translation, rotation, changement d'échelle



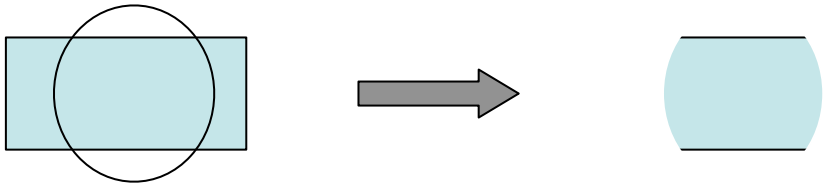
- un objet peut avoir un parent :

  ses coordonnées sont relatives à ce parent
  ici les poignées ou pour parent le rectangle et sont transformées avec celui-ci



- un objet peut avoir une forme de clipping :

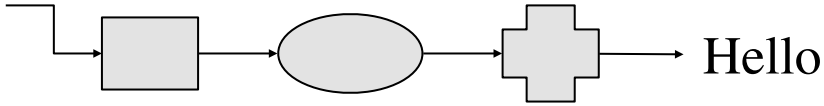
  il n'est visible qu'à l'intérieur de cette forme

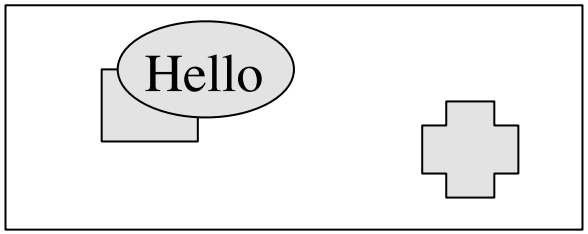


Dessin structuré : relations entre objets

Affichage dans l'ordre de la liste, « Picking » dans l'ordre inverse

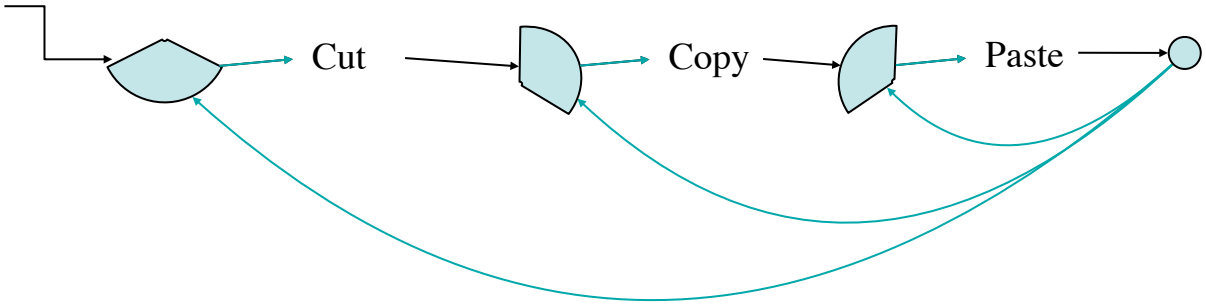
Chaque objet peut être affichable ou non, « pickable » ou non

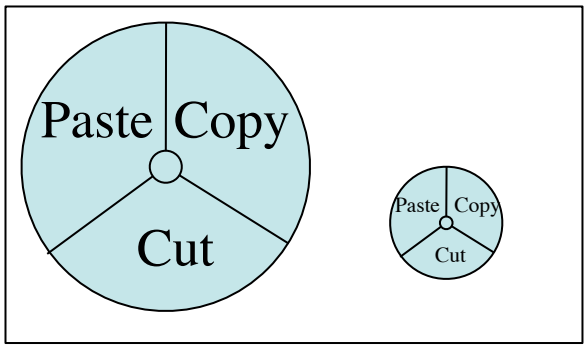




Relation hiérarchique (parent)

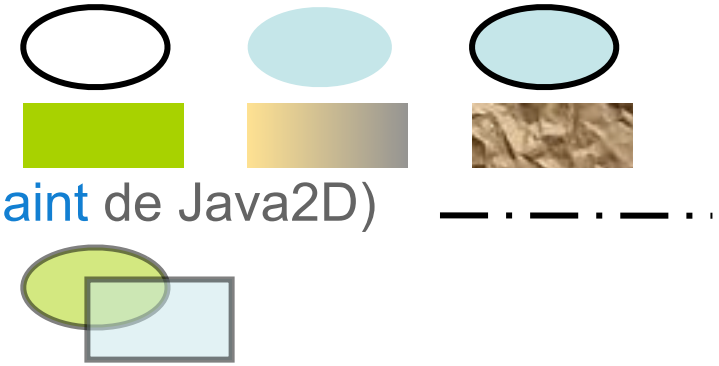
indépendante de l'ordre d'affichage :
le parent peut être avant ou après le fils dans la liste
(idem pour le clipping)





Dessin structuré : attributs graphiques

Formes géométriques :
- affichage du fond et/ou du bord
- peinture de fond (Paint de Java2D)
- forme et peinture du bord (Stroke et Paint de Java2D)
- transparence



Texte :
- police de caractères (Font de Java2D)
- couleur (Paint de Java2D)
- transparence

Hello Hello
Hello
Hello

Image :
- fichier contenant l'image
- transparence

Dessin structuré : syntaxe enchaînée

Les méthodes de SwingStates renvoient le plus souvent l'objet "this"

Ceci permet d'enchaîner les appels :

- shape = new CRectangle(...);
  shape.setFillColor(Color.RED).setStroke(new BasicStroke(1))
  .translateBy(10,100).rotateTo(45).setParent(...) ... ;

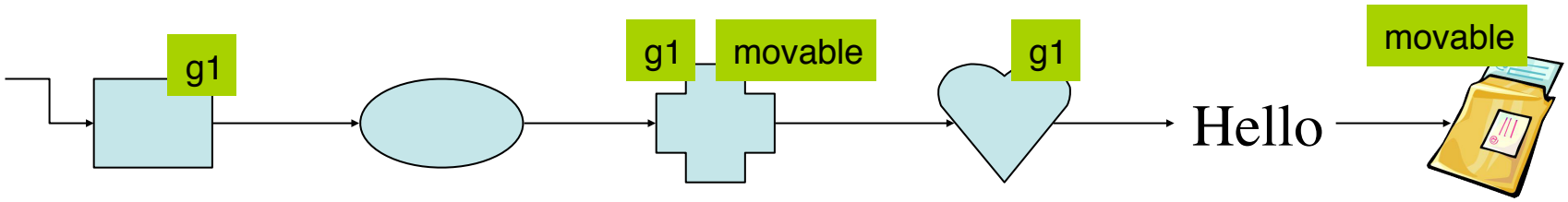
- canvas = new Canvas(...);
  shape1 = new CRectangle(...); ...
  shape2 = new CEllipse(...); ...
  canvas.addShape(shape1).addShape(shape2). ...;

- canvas = new Canvas(...);
  canvas.newRectangle(...).setFillColor(Color.RED) ...;
  Attention : ici newRectangle retourne le Shape, pas le canvas.
  Ceci permet de spécifier les attributs du rectangle à la suite.

Dessin structuré : tags

Tag = étiquette

- chaque objet peut avoir 0, 1 ou plusieurs tags
- un même tag peut être mis sur 0, 1 ou plusieurs objets



On peut appliquer aux tags la plupart des opérations que l'on peut appliquer à un objet, par exemple déplacer ou changer la couleur :

```
tag.rotateBy(45).setFillColor(Color.BLUE) ...
```

Deux rôles :

- manipuler des groupes d'objets
- définir les interactions applicables aux objets

Dessin structuré : deux types de tags

Tags extensionnels - CExtensionalTag

- posés explicitement sur chaque objet (addTag, removeTag)
- méthodes “added” et “removed” du tag appelées lorsque le tag est ajouté ou retiré d’un objet
- *le plus fréquemment utilisé : CNamedTag*

Tags intensionnels - CIntentionalTag

- définis par un prédicat (méthode “criterion”) à définir dans une sous-classe
- l’ensemble des objets qui ont le tag est calculé à chaque utilisation du tag en testant le prédicat sur chaque objet du canvas
- *potentiellement coûteux*
- *une classe utile : CHierarchyTag*
  tag intensionnel qui contient tous les descendants (par le lien parent) d’un objet

Dessin structuré : animation

Certains attributs géométriques et graphiques peuvent être animés

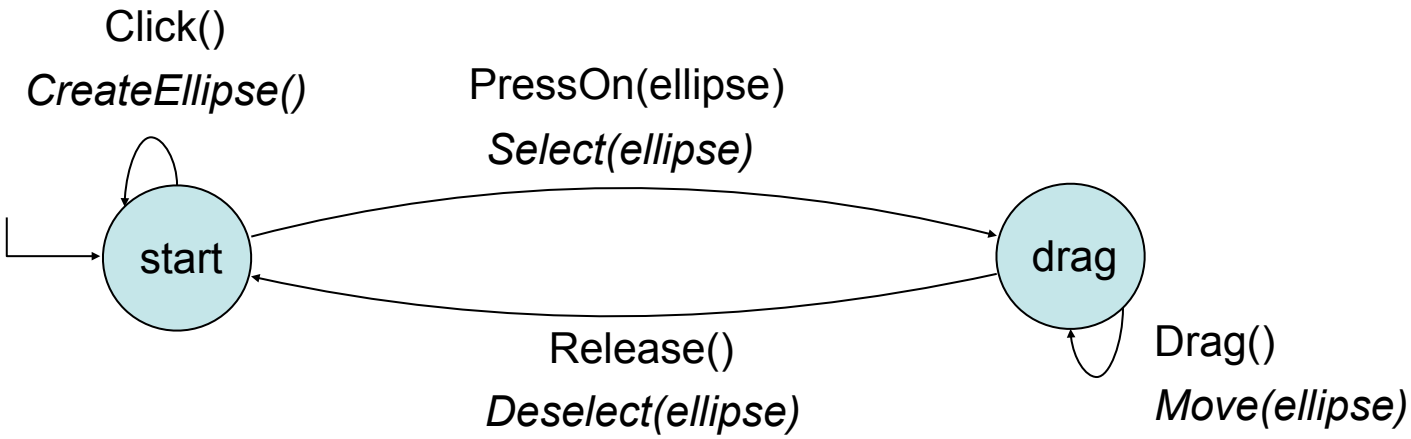
- couleur de bord et de fond
- degré de transparence
- position, taille, rotation

```
// créer une forme
rect = canvas.newRectangle(100, 150, 1, 1);
// définir l'animation
animation = new AnimationScaleTo(400, 600);
// l'associer à la forme
rect.animate(animation);
// l'animation commence immédiatement
```

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

Machines à états

Forme graphique pour la création et le déplacement d'une forme graphique



Forme textuelle idéalisée : chaque état contient ses transitions sortantes

```
Etat start {
     Transition PressOn(ellipse)  => drag {Select(ellipse)}
     Transition Click => start {CreateEllipse()}
}
Etat drag {
     Transition Drag  => drag {Move(ellipse)}
     Transition Release  => start {Deselect(ellipse)}
}
```

Machines à états : syntaxe Java

Utilisation des classes anonymes pour ressembler à la forme textuelle

```
StateMachine sm = new CStateMachine () {
   // déclarations locales
   ...
   public State start = new State ()









}
```

Machines à états : syntaxe Java

Le premier état est l'état initial

```
StateMachine sm = new CStateMachine () {
   // déclarations locales
   ...
   public State start = new State () {




   }
   public State drag = new State () {


   }
}
```

Machines à états : syntaxe Java

Utilisation des classes anonymes également pour les états et les transitions

```
StateMachine sm = new CStateMachine () {
   // déclarations locales
   ...
   public State start = new State () {
       Transition t1 = new Press (BUTTON1, "=> drag") {
              public void action () { // do something }
       }

   }
   public State drag = new State () {


   }
}
```

Machines à états : syntaxe Java

Les transitions sont testées dans l'ordre de leur déclaration

```
StateMachine sm = new StateMachine ("sm") {
   // déclarations locales
   ...
   public State start = new State () {
       Transition t1 = new Press (BUTTON1, "=> drag") {
              public void action () { // do something }
       }
       Transition t2 = ...
   }
   public State drag = new State () {
       ...
   }
}
```

Machines à états : syntaxe des transitions

```
Transition t1 = new <Transition>(<paramètres>,<état d'arrivée>) {
   public boolean guard () { ... }    // garde (optionnelle)
   public void action () { ... }      // action (optionnelle)
}
```

`<Transition>` :

Type de l'événement (Press, Release, Drag, Move, TimeOut, Key etc.) et éventuellement son contexte (OnShape, OnTag, etc.)

`<paramètres>` :

Informations supplémentaires : bouton utilisé, touche du clavier, etc.

`<état d'arrivée>` :

spécifié sous forme d'une chaîne de caractères :

"`s1`" correspond à l'état déclaré `State s1 = ...`

les caractères -, =, > et espace sont ignorés :

on peut écrire "`-> s1`" ou "`==> s1`" ou "`>> s1`", etc.

Machines à états : syntaxe des états

```
public State s1 = new State() {
   // déclarations locales si besoin
   ...
   // appelé lorsque l'état devient actif (optionnel)
   public void enter () { ... }
   // appelé lorsque l'état devient inactif (optionnel)
   public void leave () { ... }

   // déclarations des transitions
   Transition t1 = new ...
   ...
}
```

- les états doivent être déclarés public pour être visible des transitions
- les transitions peuvent accéder aux variables et méthodes de l'état englobant et de la machine à états englobante
- les états peuvent accéder aux variables et méthodes de la machine à états englobante.

Forme générale d'une application

```
class MyWidget extends Canvas {
   // déclarations locales
   ...
   // machines à états
   CStateMachine m1 = new CStateMachine () { ...}
   CStateMachine m2 = new CStateMachine () { ...}

   // constructeur
   MyWidget () {
        // créer le contenu du canvas

        newRectangle(...);
        ...
        // activer une machine à états (on peut en activer plusieurs)
        attachSM(m1, true);
   }
   ...
}
```

Forme générale d'une application

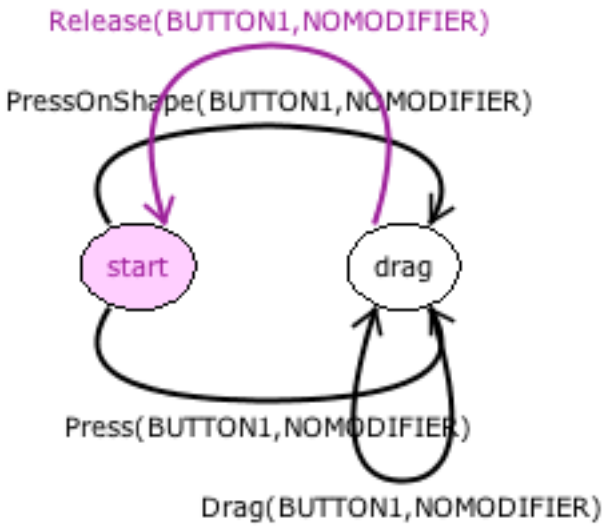
```
class MyWidget extends Canvas {

    ...

    // programme principal
    public static void main(String[] args) {
         JFrame frame = new JFrame();
         MyWidget widget = new MyWidget();
         frame.getContentPane().add(widget);
         frame.pack();
         frame.setVisible(true);
    }
}
```

Visualisation d'une machine

L'appel `new StateMachineVisualization(sm)`

crée un composant Swing contenant une visualisation graphique animée de la machine à états

- L'état courant et la dernière transition déclenchée sont colorés



Chaque machine peut également avoir des listeners qui sont déclenchés pendant le fonctionnement de la machine

Reconnaissance de gestes

SwingStates implémente deux algorithmes de reconnaissance de geste :
Rubine et $1 (Wobbrock)

L'application ***Training*** permet de créer des fichiers vocabulaire(s) de gestes

```
java -jar SwingStates.jar fr.lri.swingstates.gestures.Training
```

Construire un classifieur

```
classifier = RubineClassifier.newClassifier("classifier/cutcopypaste.cl");
classifier = Dollar1Classifier.newClassifier("classifier/cutcopypaste.cl");
```

Classifier un geste

```
String gc = classifier.classify(gesture);
```

L'écho de la trace doit être fait par l'application.
Typiquement, une machine gère le dessin de l'encre et envoie
des événements correspondants aux gestes reconnus à une autre machine.

Voir l'exemple dans le tutorial en ligne

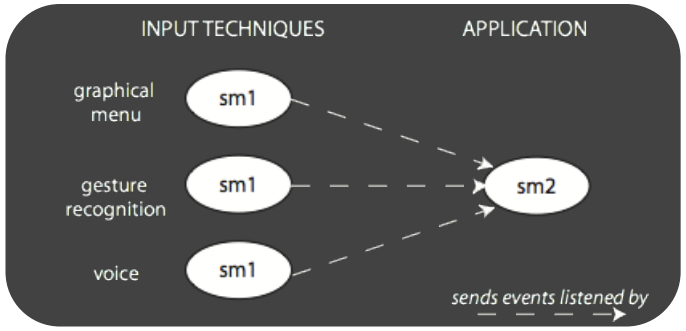
Reconnaissance de gestes

```
smGesture = new CStateMachine(c) {
  public State start = new State() {
    Transition copy = new EventOnShape("copy"){...};
    Transition cut = new EventOnShape("cut"){...};
    Transition paste = new EventOnPosition("paste"){...};
  };
};

smInk = new CStateMachine(c) {
  public State start = new State() {
    Gesture gesture = new Gesture();
    Transition begin = new Press(BUTTON1, ">> drag"){...gesture.reset();...};
  }
  public State drag = new State() {
    Transition draw = new Drag(BUTTON1){... gesture.addPoint(...); ... };
    Transition end = new Release(BUTTON1, ">> start"){...
        GestureClass gc = classifier.classify(gesture);
        if(gc != null) canvas.processEvent(gc.getName(), getPoint());
    ...};
  };
```

Contrôler l'explosion du nombre d'états

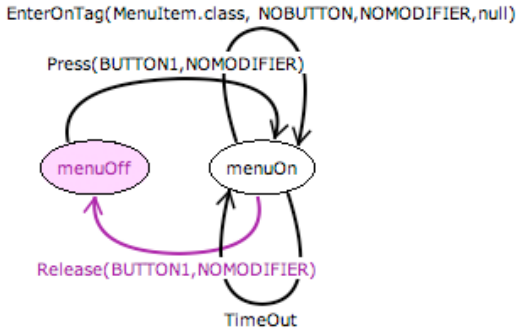
Communication entre machines

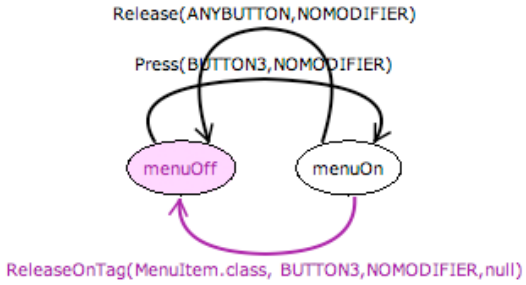
- machines de bas niveau
  pour les événements d'entrée
- envoient des événements à des
  machines de plus haut niveau



Machines en parallèle pour gérer des interactions indépendantes

- exemple : tooltips et sélection

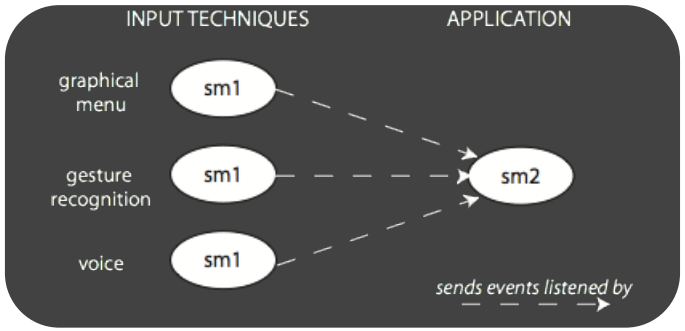




Exploiter l'héritage pour factoriser des états communs

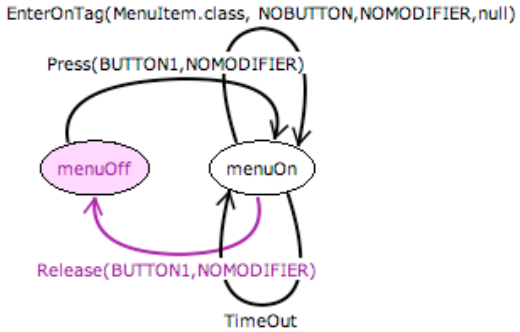
Contrôler l'explosion du nombre d'états

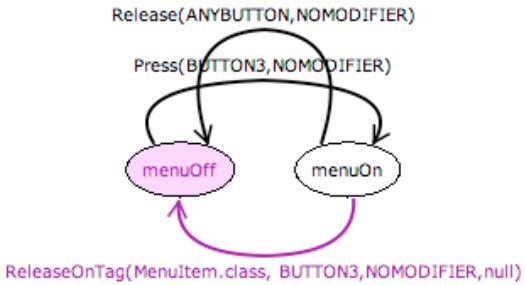
Communication entre machines



Une machine peut émettre des evts (`fireEvent`)
Une machine peut être écoutée (`addStateMachineListener`)

Machines en parallèle





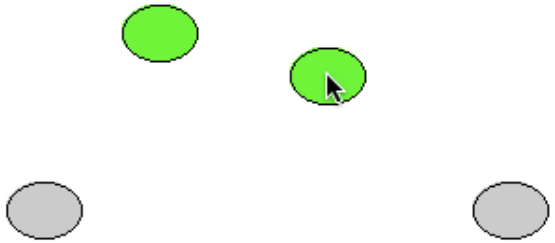
Exploiter l'héritage pour factoriser des états communs

Les tags extensionnels

- Apposés explicitement sur les objets graphiques
- Actions associées à l'ajout / retrait d'un tag

  Exemple : selection

```
CExtensionalTag selectionTag = new CExtensionalTag() {
    public void added(CShape s) {
        s.setFillPaint(Color.GREEN);
    }
    public void removed(CShape s) {
        s.setFillPaint(Color.LIGHT_GRAY);
    }
};
...
// Dans une machine à états
Transition select = new ClickOnShape(BUTTON1) {
    public void action() {
        if(getShape().hasTag(selectionTag)) moved.removeTag(selectionTag);
        else moved.addTag(selectionTag);
    }
}
```



Les tags intentionnels

- ne sont pas apposés explicitement sur les objets graphiques
- définis par un prédicat

  Exemple : selection

```
CIntentionalTag upTag = new CIntentionalTag {
   public boolean criterion(CShape s) {
         return s.getMaxY() < 500;
   }
}
```

```
upTag.translateBy(0, 50);
upTag.setFillPaint(Color.LIGHT_GRAY);
```

L'ensemble des objets "taggés" est calculé à chaque utilisation

Un événement, trois contextes

Dans SwingStates, il y a 3 types de contexte pour un même événement :

- ∅ : l'evt est survenu n'importe où (ex : Click)
- OnShape : l'evt est survenu sur une forme (ex : ClickOnShape)
- OnTag : l'evt est survenu sur une forme "taggée" (ex : ClickOnTag)

```
Transition t = new ClickOnShape(BUTTON1) {
   ...
}
```

⇔

```
Transition t = new Click(BUTTON1) {
   public boolean guard() {
      return canvas.pick(getPoint()) != null;
   }
   ...
}
```

```
Transition t = new ClickOnTag(selectionTag,
                                        BUTTON1) {
   ...
}
```

⇔

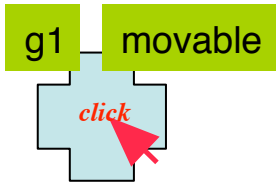
```
Transition t = new ClickOnShape(BUTTON1) {
   public boolean guard() {
      return getShape().hasTag(selectionTag);
   }
   ...
}
```

Ordre des transitions

Lors d'un événement, l'algorithme de SwingStates essaie de déclencher les transitions selon l'ordre :

- De la plus spécifique à la moins spécifique : OnTag > OnShape > ∅
- A spécificité égale, l'ordre respecté est l'ordre de déclaration

La première transition "déclenchable" est déclenchée.




```
Transition t = new ClickOnShape(BUTTON1) {
    ...
}
Transition t = new Click(BUTTON1) {
    ...
}
Transition t = new ClickOnTag("g1", BUTTON1) {
    ...
}
```

```
Transition t = new ClickOnShape(BUTTON1) {
    ...
}
Transition t = new Click(BUTTON1) {
    ...
}
Transition t = new ClickOnTag("movable",
                                        BUTTON1) {
    ...
}
Transition t = new ClickOnTag("g1", BUTTON1) {
    ...
}
```

Machines à états pour widgets Swing

- La classe JStateMachine permet de redéfinir l'interaction avec un widget Swing quelconque
- Les transitions ont le suffixe "OnComponent"
- La machine peut être attachée
  - à un widget particulier,
  - à tous les widgets de la même classe, ou
  - à tous les widgets qui ont un tag (classes JTag, JNamedTag)

```
smWidgets = new JStateMachine() {
    ...
    public State out = new State() {
        Transition enter = new EnterOnComponent(">> in") {
            public void action() {
                initColor = getComponent().getBackground();
                getComponent().setBackground(Color.YELLOW);
            }
        };
    };
    ...
};
...
smWidgets.attachTo(getContentPane());
```

Aperçu des classes

StateMachine
- CStateMachine
- JStateMachine

State

Transition

Canvas

CShape
- CRectangle
- CEllipse
- CPolyline
- CText
- CImage

Animation

CTag
- CExtensionalTag
  - CNamedTag
- CIntentionalTag

JTag
- JExtensionalTag
  - JNamedTag

Gesture

GestureClass

AbstractClassifier
- RubineClassifier
- Dollar1Classifier

Sous-classes de Transition

Click	ClickOnShape	ClickOnTag
Press	PressOnShape	PressOnTag
Release	ReleaseOnShape	ReleaseOnTag
Drag	DragOnShape	DragOnTag
Move	MoveOnShape	MoveOnTag
Enter	EnterOnShape	EnterOnTag
Leave	LeaveOnShape	LeaveOnTag

KeyPress

KeyRelease

KeyType

TimeOut