SwingStates
programming graphical interactions in Java

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

http://swingstates.sourceforge.net

Programming interactions in Java

"Listeners"

- implementation in Java of a callback function
- a "listener" objet is associated with a widget
- a method of this "listener" method is called when the widget changes state

Advantages

- simple concept
- notation facilitated by anonymous inner classes

Disadvantages

- a complex interaction gets divided among multiple listeners
- the interaction's logical path is hard to follow
- maintenance is difficult

Alternative : state machine

State machines

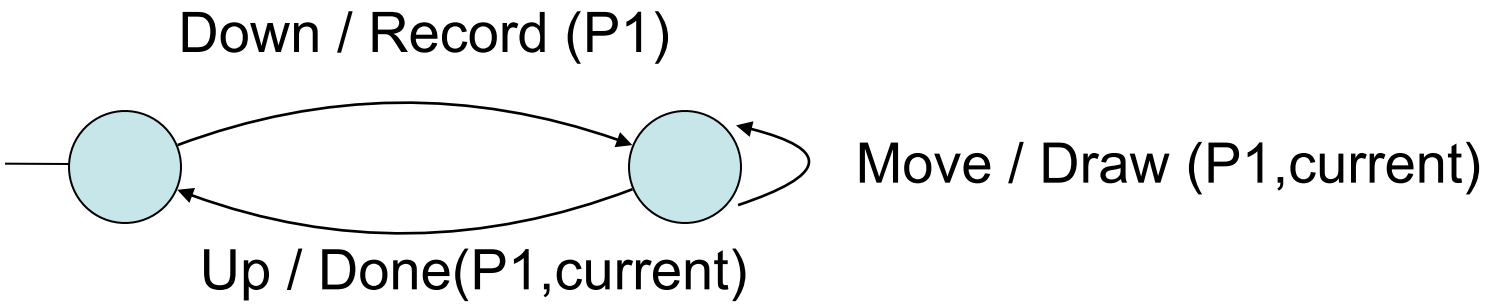
Finite state automata

- State (circle) = interaction state
- Transition (arc/link) = input events (Up, Down, Move, ...)

State machine

- actions associated with transitions (after the “/” symbol)
- guard conditions associated with transitions (after the “&” symbol)

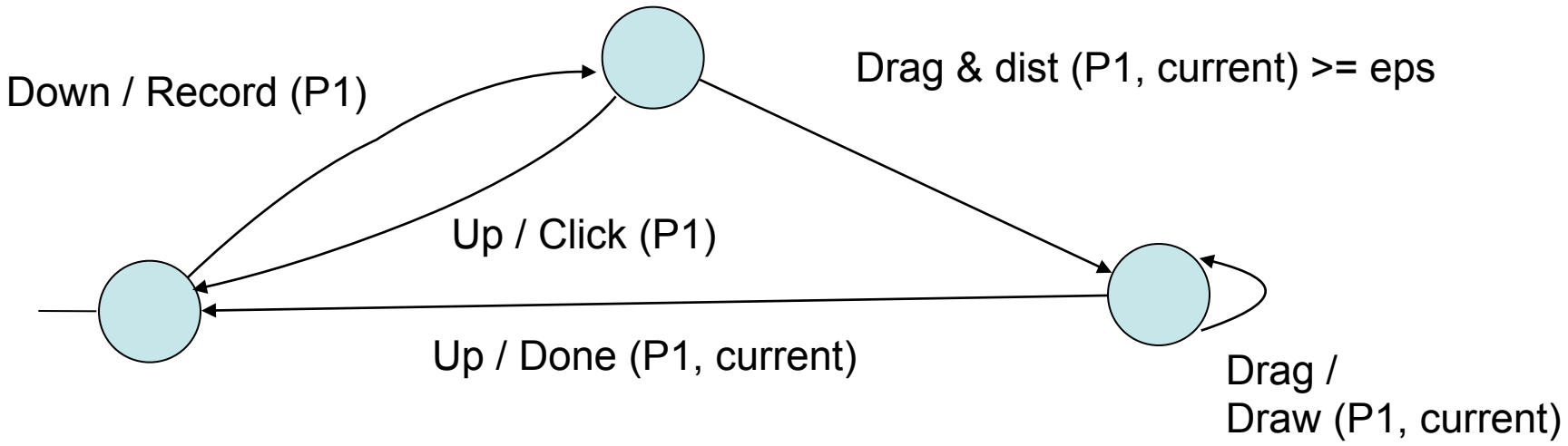
Example : "Rubber-band"



State machines

Combine selection and ink

- Hysteresis: the ink only starts after a sufficient input displacement



SwingStates : basic principles

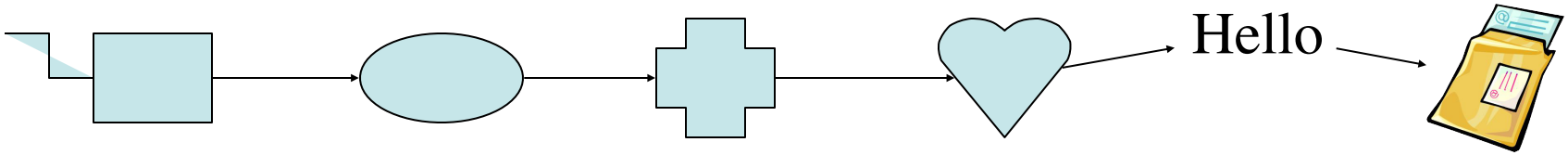
Scene graph: widget Canvas

- "display list" of graphical shapes
- graphical shapes defined by:
  - a geometry
  - graphical attributes
  - "tags"
- management of "drawing" and "picking"

State machines

- direct specification in Java of states & transitions
- associated with a Canvas
- associated with any Swing widget

Widget Canvas : scene graph



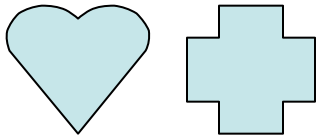
Display list:

Display objects following the list order: first object is below others

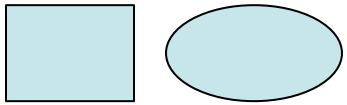
“Picking” in the inverse order, to respect the order of superposition

Geometry:

CPolyline: any shape/form



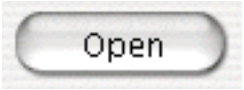
CRectangle, CEllipse: simple forms



CText: character chain Hello

CImage: image

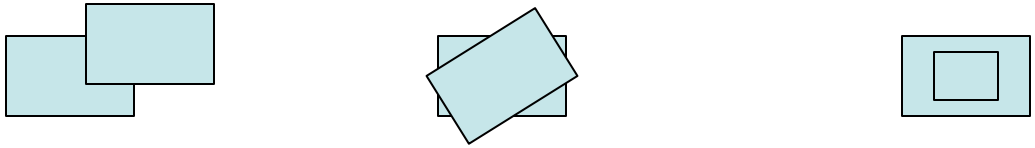
CWidget: Swing widget



Scene graph: geometry

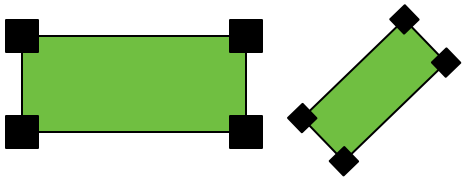
- affine transformations

  translation, rotation, scaling



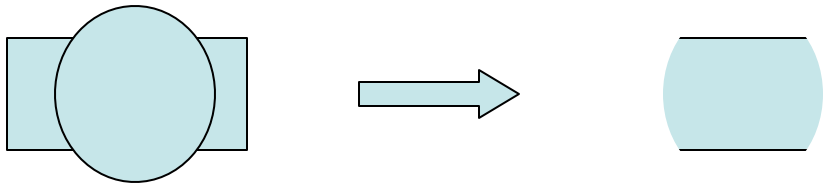
- an object can have a parent:

  its coordinates are relative to the parent
  here the handles are children of the rectangle
  and are transformed with it



- objects can have a clipping shape:

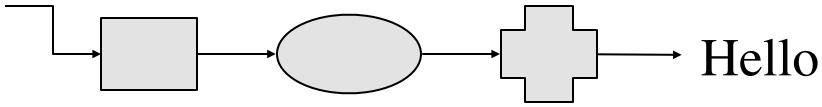
  it is visible only in the interior of the clipping shape

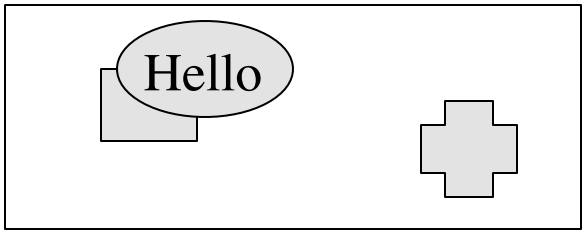


Scene graph: relationships between objets

Display in the order of the list, « Picking » in the inverse order

Each object is either « drawable » or not, « pickable » or not



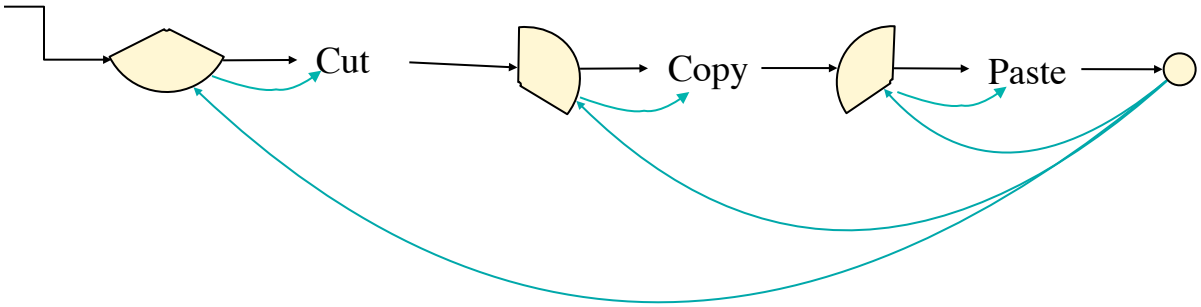


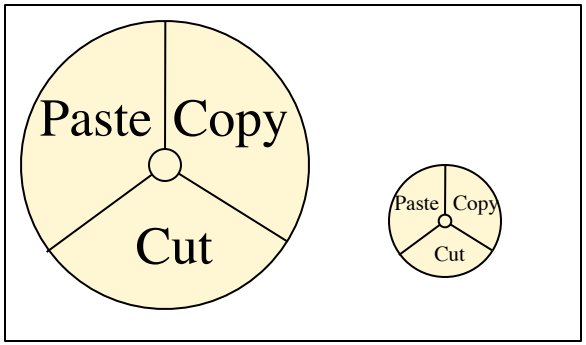
Hierarchical relationship (parent)

independent of display order:

a parent can be placed before or after a child in the list

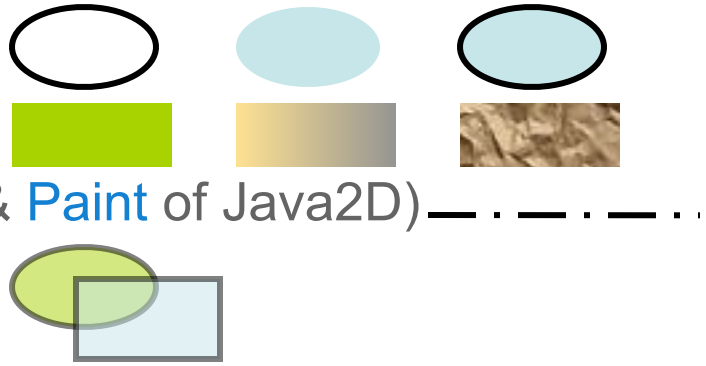
(same for clipping)





Scene graph: graphical attributes

Geometric shapes:
- display of interior (fill) and/or border
- interior fill drawing (Paint of Java2D)
- shape and drawing of border (Stroke & Paint of Java2D)
- transparency



Text:
- character fonts (Font of Java2D)
- color (Paint of Java2D)
- transparency

Hello Hello
Hello
Hello

Images:
- file containing image
- transparency

Scene graph: chain syntax

SwingStates methods most often return the object “this”

This allows chained calls:

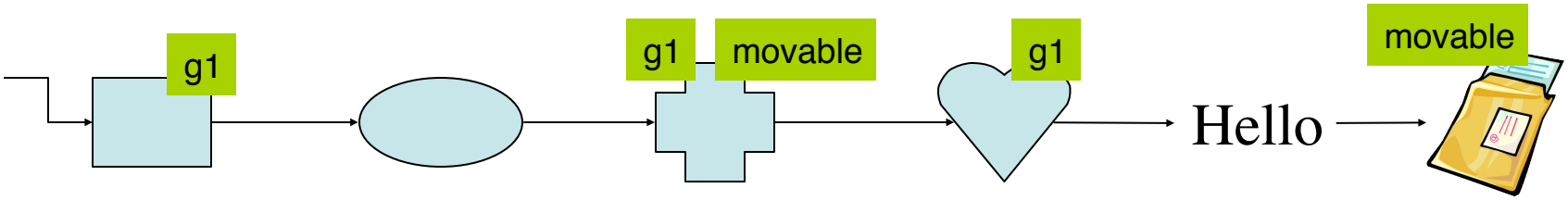
```
shape = new CRectangle(...);
shape.setFillColor(Color.RED).setStroke(new BasicStroke(1))
        .translateBy(10,100).rotateTo(45).setParent(...) ... ;

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

canvas = new Canvas(...);
canvas.newRectangle(...).setFillColor(Color.RED) ...;
```

Attention : here newRectangle returns the Shape, not the canvas.
This allows us to specify the attributes of the rectangle next.

Scene graph: tags

Tag = label

- each object can have 0, 1 or more tags
- the same tag can be placed on 0, 1 or more objects



We can apply on tags most operations we apply on objects, for example move or change color:

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

Two roles:

- manipulate groups of objects
- define interactions applicable to objects

Scene graph: two tag types

Extensional Tags - CExtensionalTag

applied explicitly on each objet (addTag, removeTag)

methods “added” and “removed” tag called
when a tag is added or removed from an objet

the most commonly used : CNamedTag

Intentional Tags - CIntentionalTag

defined by a predicate (method “criterion”)
to be redefined in a subclass

all objects that have the tag are calculated/processed at each use of the the tag, by testing the predicate against each canvas object

potentially expensive

a useful class: CHierarchyTag
intentional tag that contains all descendants of an object
(through the link “parent”)

Extensional tags

- Attached explicitly to graphical objects
- Actions associated with adding / removing a tag

  Example : selection

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

- Simple extensional tags: CNamedTag

```
titleBar.addTag("title bar");
<==>
titleBar.addTag(new CNamedTag("title bar"));
```

Intentional tags

- Not explicitly attached to graphical objects
- Defined by a predicate

  Example : selection

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

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

All canvas objects are inspected (to check if they satisfy the criterion) each time a method is called on an intentional tag

Canvas and tags

Get the list of all tags in a graphical scene

- method "getAllTags" in Canvas

Get an intentional tag grouping all shapes that have a page of the same class

- method "getTag(Class tagClass)" in Canvas

```
class Tool extends CExtensionalTag {
  protected String toolName;
  public Tool(String toolName) {
    this.toolName = toolName;
  }
}
```

```
rectangleTool.addTag(new Tool("rectangle"));
pathTool.addTag(new Tool("path"));
ovalTool.addTag(new Tool("oval"));
...
canvas.getTag(Tool.class).translateBy(...);
```

Get a given named tag

- method "getTag(String name)" in Canvas

```
canvas.getTag("title bar");
```

Scene graph: animation

Certain geometrical and graphical attributes can be animated

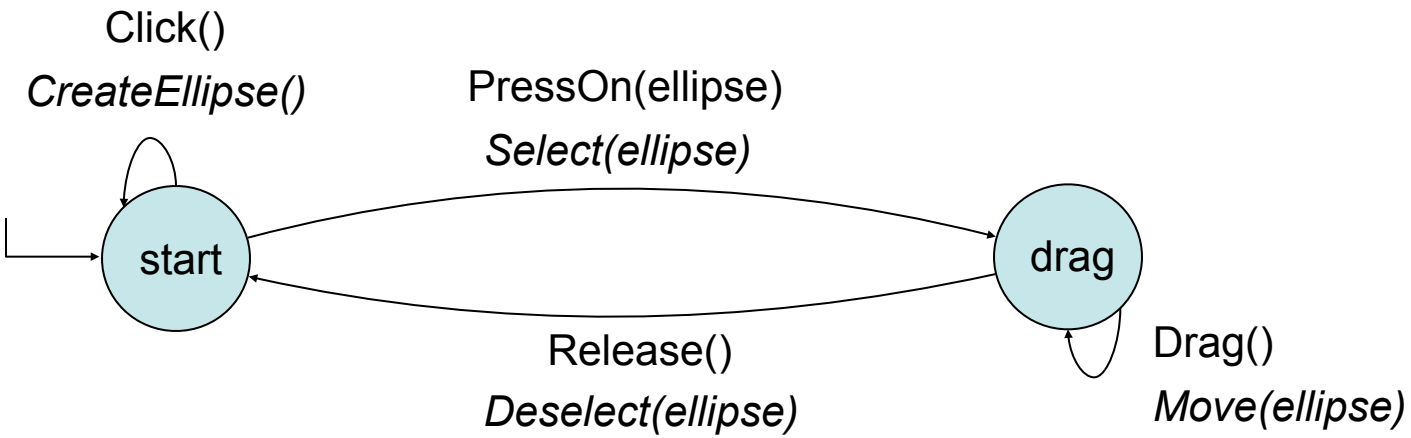
- fill and border color
- transparency level
- position, size, rotation

```
// create a shape
rect = canvas.newRectangle(100, 150, 1, 1);
// define the animation
animation = new AnimationScaleTo(400, 600);
// link it to shape
rect.animate(animation);
// the animation starts immediately
```

Other animations can be defined by extending the basic class Animation

State machines

Graphical representation for the creation and moving of a graphical object



Ideal text form: each state has its outgoing transitions

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

State machines: Java syntax

Use of anonymous classes to resemble the text form

```
StateMachine sm = new CStateMachine () {
   // local declarations
   ...
   State start = new State ()









};
```

State machines: Java syntax

The first state is the initial state

```
StateMachine sm = new CStateMachine () {
   // local declarations
   ...
   State start = new State () {




   };
   State drag = new State () {


   };
};
```

State machines: Java syntax

Use of anonymous classes also for the states and transitions

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

   };
   State drag = new State () {


   };
};
```

State machines: Java syntax

Transitions are checked in order of declaration

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

State machines: transition syntax

```
Transition t1 = new <Transition>(<parameters>,<next state>) {
   public boolean guard () { ... }    // guard (optional)
   public void action () { ... }      // action (optional)
}
```

`<Transition>` :

Type of event (Press, Release, Drag, Move, TimeOut, Key etc.) and eventually its context (OnShape, OnTag, etc.)

`<parameters>` :

Additional information: button used, key on the keyboard, etc.

`<next/arrival state>` :

specified as a string of characters:

"`s1`" corresponds to the declared state `State s1 = ...`

the characters -, =, > and space are ignored:

we can write "`-> s1`" or "`==> s1`" or "`>> s1`", etc.

One event, three contexts

In SwingStates, there are 3 context types for the same event:

- ∅ : the event occurs anywhere (e.g.: Click)
- OnShape : the event occurs on a shape (e.g.: ClickOnShape)
- OnTag : the event occurs on a "tagged" shape (e.g.: ClickOnTag)

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

⇔

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

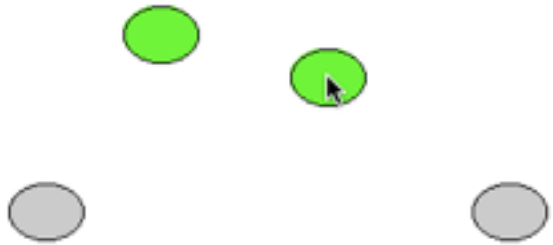
```
Transition t = new ClickOnTag(selectionTag,
                                        BUTTON1) {
   ...
}
```

⇔

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

Transitions and context

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



```
// Inside a state machine
CShape moved;
…
Transition select = new ClickOnShape(BUTTON1) {
    public void action() {
        if(getShape().hasTag(selectionTag))
            moved.removeTag(selectionTag);
        else
            moved.addTag(selectionTag);
    }
}
```

Transitions and context

```
CExtensionalTag selectionTag = new CExtensionalTag() {
    public void added(CShape s) {
        s.setFillPaint(Color.GREEN);
    }
    public void removed(CShape s) {
        s.setFillPaint(Color.LIGHT_GRAY);
    }
};
...
// Inside a state machine
Transition deselect = new ClickOnTag(selectionTag, BUTTON1) {
    public void action() {
        moved.removeTag(selectionTag);
    }
}
Transition select = new ClickOnShape(BUTTON1) {
    public void action() {
        moved.addTag(selectionTag);
    }
}
```

onTag context

Three forms of onTag transitions

- Tag object: activated when event occurs on a shape having this tag

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

- Tag class: activated when event occurs on a shape having a shape of this type/class

```
class Tool extends CExtensionalTag { ... }

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

- Tag name: activated when event occurs on a shape having the corresponding named tag

```
Transition t = new ClickOnTag("title bar", BUTTON1) {
   ...
}
```

State machines: state syntax

```
public State s1 = new State() {
// local declarations if needed
...
// called when the state becomes active (optional)
public void enter () { ... }
// called when the state becomes inactive(optional)
public void leave () { ... }

// declaration of transitions
Transition t1 = new ...
...
}
```

transitions can access variables and methods of their input states and state machines they are declared in

states can access variables and methods of the encompassing state machine

General form of an application

```
class MyWidget extends Canvas {
    // local declarations
    ...
    // state machines
    CStateMachine sm1 = new CStateMachine () { ...}
    CStateMachine sm2 = new CStateMachine () { ...}


    // constructor
    MyWidget () {
         // create the content of the canvas

         newRectangle(...);
         ...
         // activate a state machine (we can activate more than one)
         sm1.attachTo(this);
    }
    ...
}
```

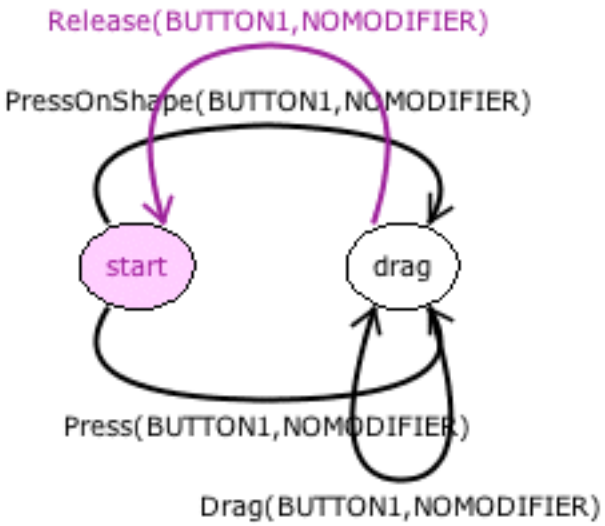
General form of an application

```
class MyWidget extends Canvas {

    ...

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

Visualization of a state machine

The call `new StateMachineVisualization(sm)`

creates a Swing component containing an animated graphic display of the state machine



The current state and the last transition triggered are colored

Each machine can also have listeners (`StateMachineEventListener`) that implement callbacks for the different operations of a machine (state changed or looped, machine paused or resumed, etc.)

Gesture recognition

SwingStates implements two algorithms for gesture recognition:
Rubine and $1 (Wobbrock)

The application ***Training*** allows us to create vocabulary file(s) of gestures

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

Build a classifier

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

Classify a gesture

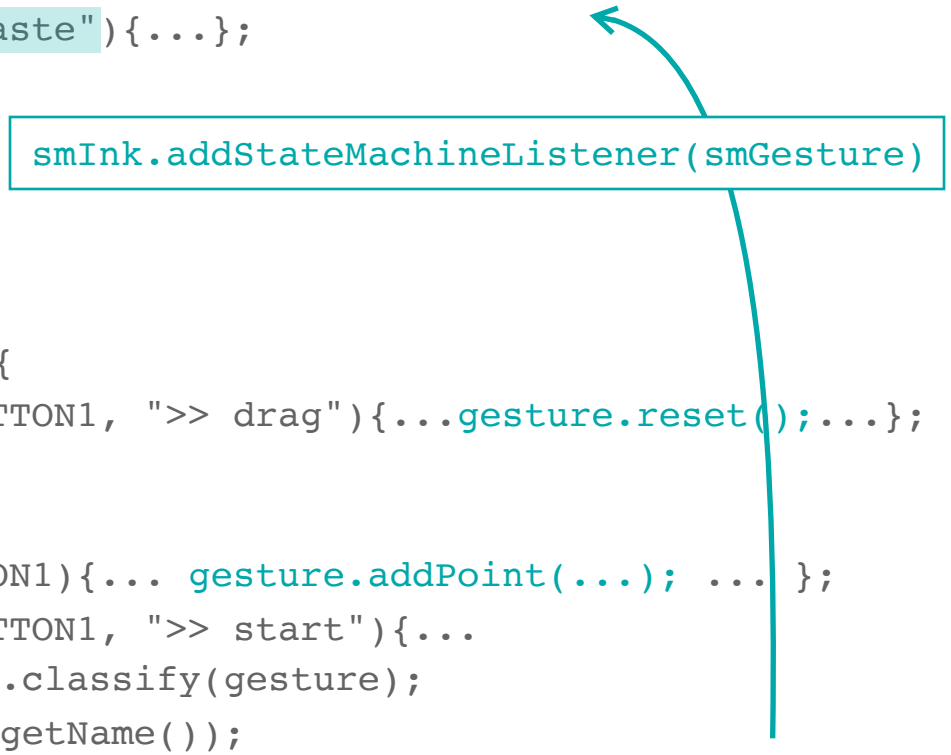
```
String gc = classifier.classify(gesture);
```

The application is in charge of handling the gesture's trace

Typically, one state machine handles the ink tracing, and sends the events that correspond to recognized gestures to another state machine.
(See detailed example in online tutorial)

Gesture recognition example 1

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

smInk = new CStateMachine(canvas) {
  Gesture gesture = new Gesture();
  public State start = new State() {
    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) fireEvent(gc.getName());
    ...};
  };
```



Gesture recognition example 1

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

We may want to know ***where*** these commands occur on the canvas

Next example shows how we can fire events directly on the canvas

```
smInk = new CStateMachine(canvas) {
  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) fireEvent(gc.getName());
    ...};
  };
```

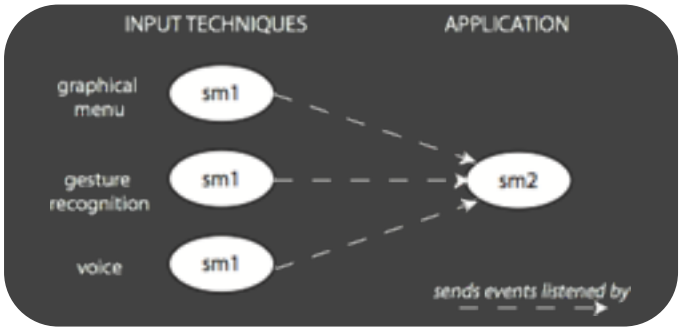
Gesture recognition example 2

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

smInk = new CStateMachine(canvas) {
  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());
    ...};
  };
```

Controlling state explosion

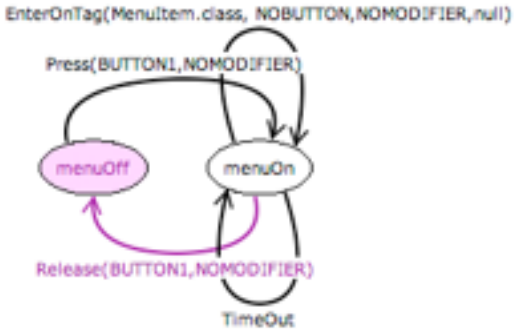
Communication between machines (stacking)

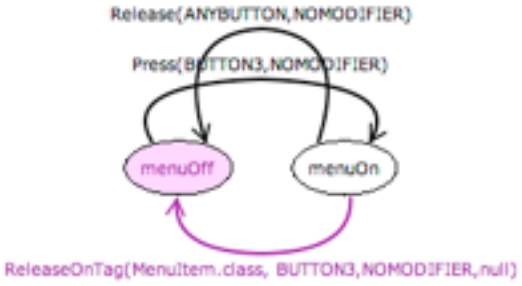
- machines at lower level for input events,
- send events to machines at higher level



Run machines in parallel to handle independent interactions

- example : tooltips and selection



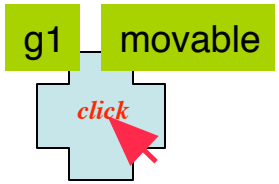


Order of transitions

When an event occurs, SwingStates triggers transitions in the following order:

From most to least specific: OnTag > OnShape > ∅

At equal specificity, the order follows the declaration order, the first matching transition is triggered.




```
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) {
    ...
}
```

State machines for Swing widgets

- The class JStateMachine allows us to redefine the interaction with any Swing widget
- The transitions have the suffix “OnComponent”
- The state machine can be attached to
  - a particular widget,
  - all widgets of the same class, or
  - all widgets that have a 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());
```

Overview of 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